ONTOLOGIES FOR INTERACTION

GERRIT NIEZEN

Enabling serendipitous interoperability in smart environments

This document was typeset using LaTeX and the typographical look-and-feel `classicthesis` developed by André Miede.

The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".

Cover design by Paul Verspaget, Verspaget & Bruinink

Printed by Eindhoven University Press

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-3234-6

Proefontwerp Technische Universiteit Eindhoven
Trefw.: Ontologies, Semantic Web, Software Architecture, Interaction Design

Ontologies for interaction

Enabling serendipitous interoperability in smart environments

PROEFONTWERP

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 9 oktober 2012 om 16.00 uur

door

Gerrit Niezen

geboren te Pretoria, Zuid-Afrika

De documentatie van het proefontwerp is goedgekeurd door de promotoren:

prof.dr.ir. L.M.G. Feijs
en
prof.dr. P. Markopoulos

Copromotor:
dr. J. Hu PDEng MEng

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

SOFIA Smart Objects For Intelligent Applications

SIB Semantic Information Broker

KP Knowledge Processor

SUMO Suggested Upper Merged Ontology

BFO Basic Formal Ontology

DUL DOLCE+DnS UltraLight

DnS Descriptions and Situations

COMM Core Ontology Multimedia

OWL Web Ontology Language

DOLCE Descriptive Ontology for Linguistic and Cognitive Engineering

FOAF Friend-Of-A-Friend

OWA Open World Assumption

BDI Belief-Desire- Intention

ODP Ontology Design Patterns

SPIN SPARQL Inferencing Notation

SWRL Semantic Web Rule Language

DC Dublin Core

RDF Resource Description Framework

URI Uniform Resource Identifier

URL Uniform Resource Locator

IoT Internet of Things

UPnP Universal Plug and Play

DCP Device Control Protocol

DLNA Digital Living Network Alliance

IP Internet Protocol

AV Audio/Video

EEG Electroencephalography

FFT Fast Fourier Transform

ANN Artificial Neural Network

REM Rapid Eye Movement

MOM Message-Oriented Middleware

JMS Java Message Service

AMQP Advanced Message Queuing Protocol

MSMQ Microsoft Message Queuing

STOMP Streaming Text Oriented Messaging Protocol

XMPP Extensible Messaging and Presence Protocol

IETF Internet Engineering Task Force

GUI Graphical User Interface

GAS Gadgetware Architectural Style

DAML DARPA Agent Markup Language

OIL Ontology Inference Layer

FSM Finite State Machine

XML Extensible Markup Language

GENA Generic Event Notification Architecture

CAMUS Context-Aware Middleware for Ubiquitous computing Systems

FIPA Foundation for Intelligent Physical Agents

UIMS User Interface Management System

UIRS User Interface Runtime System

MVC Model-View-Controller

MCRpd Model-Control-RepP-RepD

ASUR Adapter, System, User, Real object

SOUPA Standard Ontology for Ubiquitous and Pervasive Applications

CoBrA Context Broker Architecture

NFC Near Field Communication

UUID Universally unique identifier

SPARQL SPARQL Protocol and RDF Query Language

RFID Radio Frequency Identification

API Application Programming Interface

SQL Structured Query Language

PC/SC Personal Computer/Smart Card

ACS Advanced Card Systems

SSAP Smart Space Access Protocol

SAN System Architecture and Networking

QCR Qualified Cardinality Restriction

OSGi Open Services Gateway initiative

RL Rule Language

XSD XML Schema Definition

ANR Application Not Responding

CC/PP Composite Capabilities/Preferences Profile

UIML User Interface Markup Language

XIML Extensible Interface Markup Language

PUC Personal Universal Controller

INCITS/V2 URC International Committee for Information Technology Standards Universal Remote Console

RUI Remote User Interface

VNC Virtual Network Computing

RDP Remote Desktop Protocol

RFB Remote Framebuffer

UAProf User Agent Profile

DCS Distributed Communication Sphere

NORA Non-Obvious Relationship Awareness

EO Event Ontology

LODE Linked Open Descriptions of Events

TUI Tangible User Interface

MQTT Message Queue Telemetry Transport

RIBS RDF Information Base System

CWA Closed World Assumption

QR Quick Response

BIT Basic Interaction Task

OSAS Open Source Architecture for Sensors

ASP Answer Set Programming

SPICE Service Platform for Innovative Communication Environment

SLT Sound/Light Transformer

KDE Kernel Density Estimate

CDF Cumulative Distribution Function

KPI Knowledge Processor Interface

CD Cognitive Dimensions

OOP Object-Oriented Programming

RPA Relation Partition Algebra

RDFS RDF Schema

PREFACE

The work in this thesis was completed in close collaboration with another PhD candidate, Bram van der Vlist, whose thesis [116] describes the more designer-related aspects in greater detail, whereas this thesis tends to focus on the more technical aspects of the work. Some overlap between the two theses is unavoidable, but we tried to keep this to a minimum. In particular, the design iterations described in Chapters 3, 4 and 5 were a combined effort, as well as the creation of a theory of semantic connections described in Chapter 6. The device capability modelling (Chapter 7) and event modelling (Chapter 8) techniques, as well as the work on ontology engineering (Chapter 9), the extension of the ADK-SIB (Chapter 10) and the evaluations (Chapter 11) are considered to be exclusive contributions of the author.

The first person plural style of writing in the thesis is used to improve coherence and readability. Code fragments in the thesis are included either as explanation of an implementation or for the sake of clarity. The source code of the developed software[1] and ontologies[2] are available online.

A number of conference and journal papers related to this research were published in peer-reviewed proceedings and are listed on the next page.

1 https://bitbucket.org/gniezen/semanticconnections
2 https://github.com/gniezen/ontologies

PUBLICATIONS RELATED TO THIS RESEARCH

Some ideas and figures have appeared previously in the following publications:

1. Vlist, B.J.J. van der, **Niezen, G.**, Rapp, S., Hu, J., & Feijs, L.M.G. (under review). Configuring and controlling ubiquitous computing infrastructure with semantic connections: a tangible and an AR approach. *Personal and Ubiquitous Computing.* 29 pages.

2. **Niezen, G.**, Vlist, B.J.J. van der, Hu, J., & Feijs, L.M.G. (under review). Semantic Connections Theory: Enabling Interaction Designers and Developers to Create Interoperable Smart Objects. *ACM Transactions on Interactive Intelligent Systems (TiiS).* 32 pages.

3. Peeters, J. Vlist, B.J.J. van der, **Niezen, G.**, Hu, J., & Feijs, L.M.G. (2012). A Study on a Tangible Interaction Approach to Managing Wireless Connections in a Smart Home Environment. In L.- L. Chen, T. Djajadiningrat, L. Feijs, S. Fraser, S. Kyffin, & D. Steffen (Eds.) *7th International Workshop on the Design & Semantics of Form & Movement (DeSForM) 2012.* pages 187–196. Wellington, New Zealand: Koninklijke Philips Electronics N.V.

4. Vlist, B.J.J. van der, **Niezen, G.**, Rapp, S., Hu, J., & Feijs, L.M.G. (2012). Controlling Smart Home Environments with Semantic Connections: a Tangible and an AR Approach. In L.- L. Chen, T. Djajadiningrat, L. Feijs, S. Fraser, S. Kyffin, & D. Steffen (Eds.) *7th International Workshop on the Design& Semantics of Form & Movement (DeSForM) 2012.* pages 160–169. Wellington, New Zealand: Koninklijke Philips Electronics N.V.

5. Vlist, B.J.J. van der, **Niezen, G.**, Hu, J., & Feijs, L.M.G. (2012). Semantic Connections: a New Interaction Paradigm for Smart Environments. In L.- L. Chen, T. Djajadiningrat, L. Feijs, S. Fraser, S. Kyffin, & D. Steffen (Eds.) *7th International Workshop on the Design & Semantics of Form & Movement (DeSForM) 2012.* pages 16–26. Wellington, New Zealand: Koninklijke Philips Electronics N.V.

6. **Niezen, G.**, Vlist, B.J.J. van der, Bhardwaj, S. & Ozcelebi, T. (2012). Performance Evaluation of a Semantic Smart Space Deployment. *4rd International Workshop on Sensor Networks and Ambient Intelligence (SeNAmI 2012).* March 19–23, 2012, Lugano, Switzerland.

7. Hu, J., Vlist, B.J.J. van der, **Niezen, G.**, Willemsen, W., Willems, D. & Feijs, L. Designing the Internet of Things for Learning Environmentally Responsible Behaviour. *Interactive Learning Environments*, Taylor & Francis.

8. **Niezen, G.**, Vlist, B.J.J. van der, Hu, J. & Feijs, L.M.G. (2012). Using semantic transformers to enable interoperability between media devices in a ubiquitous computing environment. In Rautiainen, M. et al. (Eds.), *Grid and Pervasive Computing Workshops, Lecture Notes in Computer Science, Vol. 7096.* pages 44–53. Springer Berlin / Heidelberg.

9. Vlist, B.J.J. van der, **Niezen, G.**, Hu, J., & Feijs, L. (2011). Design semantics of connections in a smart home environment. *Creation and Design, 13*(2) p.p. 18–24.

10. Vlist, B.J.J. van der, **Niezen, G.**, Hu, J., & Feijs, L.M.G. (2011) *Interaction Primitives: Describing Interaction Capabilities of Smart Objects in Ubiquitous Computing Environments.* IEEE Africon 2011, September 13–15, Livingstone, Zambia.

11. Kwak, M., **Niezen, G.**, Vlist, B. van der, Hu, J., & Feijs, L. (2011). *Tangible interfaces to digital connections, centralized versus de- centralized.* In Z. Pan, A. Cheok, W. Mueller, and X. Yang (Eds.), Transactions on edutainment V (Vol. 6530, p. 132–146). Springer Berlin / Heidelberg.

12. Hu, J., Vlist, B.J.J. van der, **Niezen, G.**, Willemsen, W., Willems, D. & Feijs, L. (2011) *Learning from Internet of Things for Improving Environmentally Responsible Behavior.* In M. Chang et al. (Eds.), Edutainment 2011, LNCS Vol. 6872, pp. 292–299, Springer-Verlag, Berlin / Heidelberg.

13. Willemsen, W., Hu, J., **Niezen, G.** & Vlist, B.J.J. van der (2011). *Using Game Elements to Motivate Environmentally Responsible Behaviour.* Proceedings of IADIS Game and Entertainment

Technologies 2011 (GET 2011), Rome.

14. Vlist, B.J.J. van der, **Niezen, G.**, Hu, J., & Feijs, L.M.G. (2010). *Design semantics of connections in a smart home environment.* In L.-L. Chen, T. Djajadiningrat, L. Feijs, S. Kyffin, D. Steffen, & B. Young (Eds.), *Proceedings of design and semantics of form and movement (DeSForM) 2010* (p. 48–56). Lucerne, Switzer-land: Koninklijke Philips Electronics N.V.

15. Vlist, B.J.J. van der, **Niezen, G.**, Hu, J., & Feijs, L.M.G. (2010). *Semantic connections: Exploring and manipulating connections in smart spaces.* In Computers and Communications (ISCC), 2010 IEEE Symposium on (pp. 1–4). Riccione, Italy: IEEE.

16. **Niezen, G.**, Vlist, B.J.J. van der, Hu, J. & Feijs, L.M.G. (2010). *From events to goals: supporting semantic interaction in smart environments.* Proceedings of the Computers and Communications (ISCC), 2010 IEEE Symposium on, (pp. 1029–1034). IEEE.

SUMMARY

ONTOLOGIES FOR INTERACTION

ENABLING SERENDIPITOUS INTEROPERABILITY IN SMART ENVIRONMENTS

The thesis describes the design and development of an ontology and software framework to support user interaction in ubiquitous computing scenarios. The key goal of ubiquitous computing is "serendipitous interoperability", where devices that were not necessarily designed to work together should be able to discover each other's functionality and be able to make use of it. Future ubiquitous computing scenarios involve hundreds of devices. Therefore, anticipating all the different types of devices and usage scenarios a priori is an unmanageable task.

An iterative approach was followed during the design process, with three design iterations documented in the thesis. The work was done in close cooperation with designers and other project partners, in order to elicit requirements and maintain a more holistic view of the various application areas.

The thesis describes an interaction model that shows the various concepts that are involved in user interaction in a smart space, including how these concepts work together. Based in the interaction model, a theory of semantic connections is introduced that focuses on the meaning of the connections between the different entities in a smart environment.

Ontologies are formal representations of concepts in a domain of interest and the relationships between these concepts. They are used to enable the exchange of information without requiring up-front standardisation. The ontology described in the thesis helps developers to focus on modelling the interaction capabilities of smart objects and inferring the possible connections between these objects, making it easier to build smart objects and enable device interoperability on a semantic level.

Rather than just describing the low-level hardware input event that triggered an action, interaction events in the ontology are modelled as high-level input actions which report the intent of the user's action directly. This allows developers to write soft-

ware that respond to these high-level events, without having to support every kind of device that could have generated that event. The event hierarchy can be inferred using semantic reasoning.

The software architecture implements the publish/subscribe messaging paradigm, enabling smart objects to subscribe to changes in data, represented in triple form, and be notified every time these triples are updated, added or removed. Semantic reasoning is performed on an information broker, simplifying the implementation on the smart objects.

A pilot deployment, composed of heterogeneous smart objects designed and manufactured by a range of companies and institutions, was used to validate the design. A performance evaluation was performed, where the results indicated acceptable response times for a networked user interface. A usability analysis of the ontology and system implementation was performed using a developer questionnaire based on an existing usability framework. Various ontology design patterns were identified during the course of the design, and are documented in the thesis.

The resulting design artefact is an ontology for user interaction with devices in a smart environment, where devices are able to share interaction events and make use of each other's functionality.

SAMENVATTING

Dit proefschrift beschrijft het ontwerp en de ontwikkeling van een ontologie en software raamwerk ter ondersteuning van gebruikersinteractie in ubiquitous computing scenario's. De kern van ubiquitous computing is "serendipitous interoperability", waarbij apparatuur die niet noodzakelijkerwijs ontworpen is om samen te werken, in staat zou moeten zijn om elkaars functionaliteit te ontdekken en te gebruiken. In toekomstige ubiquitous computing scenario's zijn honderden apparaten betrokken. Daarom is het op voorhand overzien van de verschillende types apparaten en gebruikersscenario's een onhanteerbare opgave.

In het ontwerptraject is een iteratieve aanpak toegepast, waarvan er drie ontwerpiteraties in het proefschrift gedocumenteerd

staan. Het werk is gedaan in nauwe samenwerking met ontwerpers en andere projectpartners, om ontwerprichtlijnen te extraheren en een holistische kijk op de verschillende toepassingsgebieden te waarborgen.

Het proefschrift beschrijft een interactiemodel dat de verschillende concepten met betrekking tot gebruikersinteracties in een intelligente omgeving laat zien, inclusief hoe deze concepten onderling samenwerken. Gebaseerd op het interactiemodel, is een theorie geïntroduceerd die zich richt op de betekenis van de verbindingen tussen de verschillende entiteiten in een intelligente omgeving.

Ontologieën zijn formele beschrijvingen van concepten in een bepaald interessegebied en de verhoudingen tussen deze concepten. Ze worden gebruikt om informatie uit te wisselen zonder een voorafgaande standaardisatie te vereisen. De ontologie beschreven in dit proefschrift helpt ontwikkelaars zich te richten op het modelleren van de interactiemogelijkheden van intelligente objecten en het uitzoeken van de mogelijke verbindingen tussen de objecten, wat het vergemakkelijkt om intelligente objecten te ontwikkelen en mogelijk maakt de apparaat-interoperabiliteit op niveau van semantiek te benaderen.

In plaats van het beschrijven van een invoergebeurtenis via de hardware, die op een laag niveau een actie in werking zet, zijn de interactiegebeurtenissen in de ontologie omschreven als input actie op een hoger abstractie niveau waarbij de intentie van de gebruiker direct gerapporteerd wordt. Dit maakt het mogelijk voor ontwikkelaars om software te schrijven die reageert op de input acties op hoog niveau, zonder ondersteuning te vereisen voor alle mogelijke typen apparaten die de actie gegenereerd zouden kunnen hebben. De hiërarchie van gebeurtenissen kan automatisch worden afgeleid met behulp van semantische beredenering.

De software architectuur implementeert het zogenaamde publish/subscribe communicatie model, waarbij intelligente objecten veranderingen in data kunnen aanmelden, welke is weergegeven in triple formaat, en een notificatie ontvangen, iedere keer dat de triples zijn bijgewerkt, toegevoegd of verwijderd. De semantische beredenering wordt uitgevoerd door een informatieonderhandelaar (information broker), wat de implementatie voor de intelligente objecten vergemakkelijkt.

Een testopstelling, samengesteld uit ongelijksoortige intelligente objecten ontwikkeld door verscheidene bedrijven and instituten, is gebruikt voor validatie van het ontwerp. De prestaties van de implementatie zijn geëvalueerd, met als resultaat een indicatie van een acceptabele reactietijd voor een genetwerkte gebruikersinterface. Een gebruikersanalyse van de ontologie en systeemimplementatie is uitgevoerd met ontwikkelaars, waarbij een vragenlijst is gebruikt die gebaseerd is op een bestaand bruikbaarheidsraamwerk. Uiteenlopende ontwerp patronen voor ontologieën zijn geïdentificeerd tijdens ontwerpproces, en zijn gedocumenteerd in het proefschrift.

De resulterende uitkomst is een ontologie voor gebruikersinteractie met apparaten in een intelligente omgeving, waar apparaten de mogelijkheid hebben om interactiegebeurtenissen te delen en gebruik te maken van elkaars functionaliteit.

Part I

FRAMING THE PROBLEM AND CURRENT STATE-OF-THE-ART

In this first part of the thesis, we introduce the problem that was addressed. Related work is also described, showing the different aspects of the work that were considered.

1 INTRODUCTION

The real problems going forward are not with any single device, but in the potential complexity of the larger ecosystem of technologies that we function in. [...] It's about the society of appliances and how they work today which is the new frontier.

— Bill Buxton [20], HCI researcher and designer

Parts of this chapter appear in [80] and [81].

When trying to share music or photos between your mobile phone and that of a friend, making the connection between the two devices is not always straightforward. You need to select the appropriate communication technology and identify the target device from a list of possible options. Manufacturers often only allow sharing between devices that form part of their own ecosystem, where devices from other manufacturers may be incompatible with this ecosystem. In the future there could be hundreds of devices in your immediate surroundings, and these devices, created by different manufacturers, will need to work with another.

What if we want to share more than just media, or want to use one device to control another? As an example, imagine connecting a sleep monitor to the lamp on your bedside table, helping you to wake up at the right time in your sleep cycle. This thesis focuses on ways to create meaningful connections between devices, based on the functionality of the devices. We develop techniques for designers and developers to describe the capabilities of devices, such that the content and functionality can be shared with other devices. We also make use of different types of feedback to indicate what the possibilities for interaction are, as well as making the events that occur within this system of networked objects more transparent.

Mark Weiser [122] coined the term ubiquitous computing, sometimes seen in its shortened form as "ubicomp".

Key to realising the vision of ubiquitous computing [122] is "serendipitous interoperability", where devices which were not necessarily designed to work together should be able to discover each other's functionality and be able to make use of it [2]. Future ubiquitous computing scenarios involve hundreds of devices, appearing and disappearing as their owners carry them from one room or building to another. Therefore, anticipating all the different types of devices and usage scenarios a priori is an unmanageable task.

Next to serendipitous interoperability, another enabling strategy of ubiquitous computing is to make technologies — as from a user's perspective they are still dealing with technologies — disappear, and "weave themselves into the fabric of everyday life until they are indistinguishable from it" [122]. To reach this goal, self-configuration of the various devices and technologies in ubiquitous computing environments is essential. Whether automated and initiated by context-aware entities, or initiated by users by connecting the devices to one another, the actual configuration of the various components at a lower level should happen automatically.

1.1 BACKGROUND

1.1.1 *Multi-device user interaction*

As computers disappear into the environment, we will need new kinds of human-computer interactions to deal with the peculiarities of these smart environments, which include invisible devices, implicit interaction, and distinguishing between physical and digital interactions [128]. In the conventional Graphical User Interface (GUI) genre, designers have typically developed prepackaged solutions for a predetermined interaction space, forcing users to adapt to their specific interaction protocols and sequences. In ubiquitous computing environments, the interaction space is unpredictable and emerges opportunistically [26]. There is the risk of creating a mismatch between the system's model of interaction and the user's mental model of the system. In these conditions, new interaction techniques must be devised to help users to construct helpful mental models, in order to minimise system and user model mismatches. These interaction techniques should also match the context of use that is dynamic and unpredictable.

If we are able to connect smart devices to one another effortlessly, it becomes possible to support high-level services, that would usually involve multiple steps on multiple devices [97]. From a user's point of view, streaming music from a mobile device to a home entertainment system is a single high-level task. In practice there are multiple steps involved, and if the devices involved are from different manufacturers, the user needs to learn the operational details of each device interface in order to perform the task. From a technical perspective Universal Plug and Play (UPnP) with its device control protocols [114] is not considered an adequate solution, because it only provides static device description documents and covers a very limited number of use cases.

At home the average person interacts with many devices during the course of a day. Sometimes these devices are used by more than one person, or one device may be used as an interface to another. As these devices are manufactured by different companies, there exist many different user interfaces that must be studied before they can be used. There might even be more than one way to interact with a single device. For example, to turn down the volume on a home entertainment system, either a remote control or a volume dial on the entertainment system itself may be used. It is expected that in future, more generic tools will be used to discover, configure, connect and control all the devices in the environment [76].

1.1.2 *Configuring connections between devices*

In a world where we are potentially surrounded by a multitude of devices, allowing for the arbitrary ad hoc interconnection of devices, and the sharing of information between these devices, is difficult. It is unreasonable to expect that a device will have prior knowledge of all the different ways it can interact with surrounding devices. The number of possibilities are too large, and anticipating the potential number of interactions is infeasible. If we could add meaning to the interactions and interconnections in such a way that it is machine-readable, semantic web technologies could be used to infer additional properties about the existing entities. This could fill the gaps between that which is described in terms of device capabilities, and that which is possible in terms of combined functionality. The user is still the final arbiter in deciding what the device does, but the device should be capable of communicating the possibilities based on what was inferred from its environment.

Besides the technological challenges, there also lies a challenge ahead for designing user interactions with these ecosystems of interconnected devices. When moving away from interaction with a single device towards interactions with systems of devices, designers need to communicate the relationships between the devices, and the larger system they are part of. Additionally, designers need to find ways to communicate the action possibilities of new, "emergent functionalities" [44], that emerge when devices are being interconnected.

An important problem that arises when designing for these systems of interactive objects is their highly interactive and dynamic nature [44]. The inherent ever-changing nature of these systems and the severely limited overview of the ecosystem in its entirety is one of the most important challenges a designer faces when designing for such systems. Additionally, such a system comprises many different "nodes" that the designer, at the time of designing has no control over. Yet, when designing and adding new nodes to the system, making them interoperable is crucial for success.

According to Newman et al. [76], the following should be communicated to a user attempting to interact with and establish connections between devices:

- What devices and services are available
- Capabilities of the devices and services
- Relationships between each another and the environment
- Predictions of likely outcomes from interaction

The information presented to the user should be filtered dynamically, based on the user's context. This context includes for example the user's location, interaction history, and current tasks. A smart object is able to sense the context of its surroundings, make use of this context and other information to proactively interact with users and other smart objects, and self-organise into groups with other devices [100]. This context information should be represented in such a way that is understood by all the entities in the system.

The background described in this section provides for interesting design challenges and research questions that can be asked. In the following section we will first discuss the context of the work described in this thesis, followed by the research questions that were addressed.

1.2 CONTEXT OF THE WORK AND RESEARCH QUESTIONS

The work described in this thesis was completed as part of a European research project called Smart Objects For Intelligent Applications (SOFIA)[1]. Some of the design choices were guided by collaboration with partners in the SOFIA project. We worked

1 http://www.sofia-project.eu/

with the project partners to elicit requirements and expose ourselves to other application areas, in order to gain a more holistic view of the problem.

1.2.1 *The SOFIA project*

SOFIA is an European research project within the ARTEMIS framework that attempts to make information in the physical world available for smart services — connecting the physical world with the information world. The goal is to enable cross-industry interoperability and to create new user interaction and interface concepts, to enable users to benefit from smart environments. The centre of the software platform developed within SOFIA is a common, semantic-oriented store of information and device capabilities called a Semantic Information Broker (SIB). Various virtual and physical smart objects, termed Knowledge Processors (KPs), interact with one another through the SIB. The goal is that devices will be able to interact on a semantic level, utilising (potentially different) existing underlying services.

The SOFIA software platform is based on the ideas of space-based computing. A tuple space is a repository of tuples, where a tuple is an ordered list of elements. Tuple-based computing has been introduced in parallel programming languages to implement communication between parallel processes [41]. Producers send their data as tuples to the space, and consumers read tuples from the space. This is also known as the blackboard metaphor.

Our focus within the SOFIA project was on the user interaction aspects of devices in the smart home environment. While most of the examples in this thesis are specific to the smart home environment, the concepts are also applicable in the wider context of ubiquitous computing, for example the smart city or smart personal spaces. We now consider the context of the work in terms of the vision of ubiquitous computing.

1.2.2 *Ubiquitous computing*

The vision of ubiquitous computing describes a future where electronic devices are so ubiquitous that their presence is not noticed anymore. As described earlier in this chapter, we consider the enabling technological strategies of ubiquitous computing to be serendipitous interoperability and making technologies disappear.

Chalmer and MacColl [23] questioned the more recent assumption in ubiquitous computing research that devices should disappear into the environment, reiterating Weiser's original vision that tools for interaction should be "literally visible, effectively invisible". There is a difference between physical and cognitive disappearance. Where the term physical disappearance describes the trend of devices to be embedded into the environment, cognitive disappearance means that the user does not distinguish between the computer and the artefact anymore, but focuses on the function of the artefact. Devices should retain their unique characteristics, even when placed within systems of devices. Users are influenced by how they perceive devices, and we have to accept that the devices themselves are part of the user's context.

Ubiquitous computing products are a combination of hardware, software and services. It is not clear what kind of skills are required to design for this kind of environment [66]. There is, however, a need for interaction designers and software developers to have a common vocabulary and framework when cooperating to create these products. This thesis attempts to move this idea forward, by defining common concepts that are prevalent in most ubiquitous computing environments, and establishing a framework that can be used by both designers and developers alike.

1.2.3 *Affordances*

In their article "At Home with Ubiquitous Computing: Seven Challenges", Edwards and Grinter [37] describes a scenario where a couple come downstairs in the morning intent on listening to the radio, and realise that there is no sound coming from their speakers. It turns out that the neighbours bought a new set of Bluetooth-enabled speakers which, when installed, associated themselves with the nearest sound source – the couple's Bluetooth-enabled stereo.

The wireless nature of the speakers does away with the traditional affordances making connections between the speakers and the stereo. These affordances are explicit when physical wires are used - the connections can be observed and the range of connectivity is clear. Edwards and Grinter state that the design challenge is to provide *affordances* that help users understand the technology, allowing them to control, use and debug technologies that interact with one another in the environment.

Norman [85] defined affordances as the perceived and actual properties of an object, primarily those properties that determine how an object should be used. The set of action possibilities of an object is based on its appearance, but also on the actual interaction with that object. An affordance is a relationship between an object and the person acting on the object, such that the same object might have different affordances for different individuals. The term was originally created by the psychologist J.J. Gibson to describe human perception [50], but was extended by Norman for its application to design.

1.2.4 *Ontologies*

The current state of ubiquitous computing is similar to that of desktop computing in the 1970s, where there is a whole range of new technologies without metaphors to communicate how they operate. The question then becomes how we then can model a device, not only in terms of its technical characteristics or capabilities, but also in terms of user interaction and feedback, where metaphors, functionality and affordances play an important role.

One possible solution to modelling devices is to make use of ontologies, a concept in computer science most often associated with the Semantic Web [12]. Ontologies are formal representations of knowledge, consisting of various entities that are related to one another. They provide a shared vocabulary, which makes it easier to publish and share data. Ontologies allow us to model a domain in terms of its concepts, and the relationships between these concepts. They are also both machine-readable and human-understandable.

Ontologies are well suited to environments with a large number of devices. They have been designed to work at Web scale, they enable heterogeneous data sources to interoperate with one another, and they are based on technology standards which allow for easy and large scale adoption [100].

Ontologies lend themselves well for describing the characteristics of devices, the means to access such devices, and other technical constraints and requirements that affect incorporating a device into a smart environment [2]. Using an ontology also simplifies the process of integrating different device capability descriptions, as a semantic inferencing engine can be used to infer relationships between the concepts in the descriptions.

1.2.5 *Research questions*

The hypothesis of this thesis is that *user interaction in a smart environment can be better supported by ontological models than with existing device and service descriptions* (e.g. descriptions stored in relational databases). These ontological models define a semantic mapping between the user's behaviour and the available resources in the environment.

"The greatest challenge to any thinker is stating the problem in a way that will allow a solution." – Bertrand Russell

The thesis aims to answer a number of research questions. In the previous section, ontologies were offered as a potential solution to solving the interoperability problem in ubiquitous environments. They are also well suited to describing user interaction in such an environment. This leads us to the first question:

Research question 1. *How can we use an ontology to model user interaction and devices in a smart environment consisting of multiple devices and multiple interactions?*

Related work on ontologies [24, 94, 77], described in more detail in the next chapter, focused mainly on modelling context and basic device properties. The ontologies created as part of the work described in this thesis is not only an attempt to model device capabilities in more detail, but to our knowledge is also the first attempt to model user interaction in a smart environment.

In the SOFIA project, KPs communicate with a message broker using the blackboard architectural pattern, where the message broker contains a common knowledge base. This knowledge base, consisting of a triple store and an ontology, is used to share information between the various knowledge sources.

Research question 2. *How suitable is the blackboard architectural pattern for handling ontology-based ubiquitous computing environments?*

Suitability is defined in terms of user interaction, where the performance of the combination of a triple store, semantic reasoning and the blackboard architectural pattern is evaluated in terms of responsiveness.

The advantage of a blackboard architecture is that it decouples reference, time and space [41]. Communicating processes do not need to explicitly know each other, i.e. reference-wise they are decoupled. The blackboard guarantees persistent storage, such that communication can be asynchronous and decoupled time-wise. As long as they have access to the same blackboard, processes can run anywhere, decoupling them space-wise. We can compare the performance of our implementation against related work on blackboard-based architectures for ubiquitous computing environments [126, 38].

If we make use of a triple store and ontology, a semantic reasoning engine is required to perform inferencing on the knowledge base of asserted triples. If triples are frequently inserted and removed, the time required for inferencing could have an adverse effect on the responsiveness of the user interface.

Research question 3. *How responsive is a networked user interface that is implemented on top of a system architecture with a semantic reasoning engine?*

Some work has been done to measure the usability of Application Programming Interfaces (APIs) for developers [98]. However, we do not have a way to evaluate the usability of software frameworks and ontologies for ubiquitous computing environments from a developer point-of-view.

Research question 4. *How can we measure the usability of ontologies and software frameworks for developers of ubiquitous computing environments?*

Feedback is required to help the user make sense of what is happening in the environment. When we consider multiple interconnected smart objects, feedback and feedforward gets spatially distributed.

Research question 5. *How should feedback be provided in a networked user interface consisting of multiple connected devices?*

1.3 METHODOLOGY

An iterative design methodology [68] was followed for the work described in this thesis. This cyclic process, shown Figure 1, consists of three steps – design, implementation and evaluation – where the results of the previous iteration are used as input for the next iteration. Iterative design is commonly used in the development of human-computer interfaces, but applies to many fields, including industrial design and software engineering.

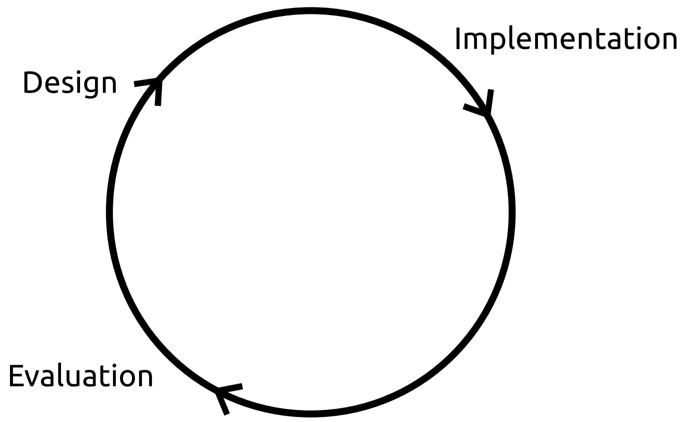


Figure 1: Iterative design methodology

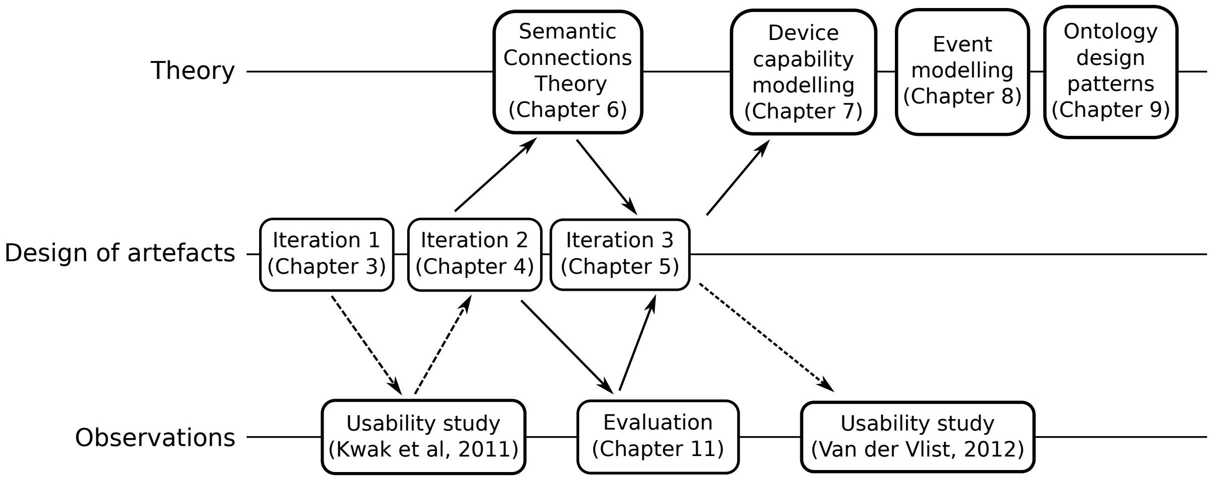


Figure 2: The research approach used in the thesis

The research described in this thesis can be aligned to the triangulation framework of Mackay and Fayard [71], as shown in Figure 2. This framework integrates scientific and design models, by combining the deductive and inductive approaches with the design approach. With the deductive approach, research originates from theory, whereas with the inductive approach the research originates from observations. With the design approach, designers and engineers use guidelines and requirements as input to iterative prototypes used to construct a final product.

The framework shows how the design iterations are related to the constructed theories and models, as well as the evaluations and observations. The framework provides a roadmap of the work described in this thesis, where the relevant chapters are indicated in the figure.

1.4 OUTLINE OF THE THESIS

In the remainder of Part A the related work, including relevant research projects, is discussed. Existing state-of-the-art ontologies for ubiquitous computing environments and context-aware systems are described, followed by a description of the various interaction models, task models and semantic models that were used as basis for our own interaction model.

An iterative approach was followed during the design process. Part B describes the three design iterations, detailing the requirements, design, implementation and evaluation processes. A theory of semantic connections is introduced, based on the output from the design iterations, that focuses on the meaning of the connections between the different entities in a smart environment. It is intended to enable interaction designers and developers to create interoperable smart objects, providing them with a common vocabulary and framework. The SOFIA software architecture was taken as a departure point during each of the design iterations described in Part B. The various extensions and changes to the reference architecture are described in more detail in each design iteration description.

From the work done, concepts and techniques that can be applied to ubiquitous computing in general were discovered. These concepts and techniques were extracted from the design iterations and are discussed in more detail in Part C, to exist independently of the design iterations. Our approach to modelling the interaction capabilities of smart objects is described, which builds on earlier ontologies for context-aware systems. Another contribution of this thesis is in the way interaction events are modelled, utilising existing event modelling techniques to describe user interaction in smart environments. Ontology design patterns that were identified and used during the course of the design are documented. A proposed software architecture to be used in future ubiquitous computing scenarios, based on the work done within SOFIA, is described. This is followed by an evaluation of the work, which includes a performance evaluation and usability analysis.

2 RELATED WORK

The old computing was about what computers could do; the new computing is about what users can do. Successful technologies are those that are in harmony with users' needs. They must support relationships and activities that enrich the users' experiences.

— Ben Shneiderman [111], computer scientist

In this chapter we describe related work. The work can be related in several ways, depending on which aspect of the work is considered. Therefore the chapter is subdivided into five sections, each devoted to one aspect. These are related projects and frameworks (Section 2.1), ubicomp ontologies (Section 2.2), user interface software architectures (Section 2.3), modelling input devices (Sections 2.4) and semantic models (Section 2.5).

2.1 RELATED PROJECTS AND FRAMEWORKS

In the field of ubiquitous computing there are a substantial number of past and current projects and relevant software frameworks that exist, most of them in the area of context-aware computing. In the following sections we will focus on those projects that are the closest in scope to the issues that are addressed by the work described in this thesis:

- Serendipitous interoperability, as addressed by the *recombinant computing* approach of the SpeakEasy project
- Sharing information between devices, as addressed by the EventHeap shared event system and its *tuple space protocol*
- Using one device to control another, as addressed by the *opportunistic assemblies* of the XWeb architecture
- Multi-device user interaction, as addressed by the *Media Cubes* of the AutoHAN project
- Configuring connections between devices, as addressed by the *plug-synapse model* of the e-Gadgets project

2.1.1 *SpeakEasy (circa 2000-2003)*

We cannot expect all devices to have a priori knowledge of all the other devices they might possibly be connected to. We can, however, expect users to have knowledge about the devices they might encounter in their environment. Even if my smart phone does not know how to communicate with a specific printer, the software infrastructure could provide the necessary technical building blocks to allow them to communicate. The user understands what a printer does and makes the decision to connect the smart phone to the printer, as well as what to print.

This line of thinking was a starting point for Newman et al [76], who developed an approach which they named *recombinant computing*, used in the SpeakEasy project at Xerox PARC. With this approach components are designed with the thought that they may be used in multiple ways, under different circumstances and for different purposes. Components expose *recombinant interfaces* that are simple, domain-independent programmatic interfaces governing how components can interoperate with one another.

The SpeakEasy project focused on what users might be trying to accomplish in a specific situation when connecting entities to one another. Possible examples include connecting devices in order to give a presentation, or in order to send contact information. They created templates of common tasks that contained a partially specified set of connections and entities, which could be fully specified and instantiated by users at run-time. An instantiated template was then added to a list of current tasks. It was noted that templates impose a layer of semantics on top of the raw infrastructure. Templates assisted users by constraining the available component choices to only those that were appropriate for the task at hand. For example, a template could be partially specified as

- Task name: Give a presentation
- Connection: File -> Projector
- File name: Choose..
- Projector: Choose..

where the user can then specify the name of the file, for example a Powerpoint presentation, as well as the projector in the room from a list of possibilities.

The SpeakEasy environment consisted of a web application that allows users to browse for components, which can be viewed and organised in different ways, for example grouped by location or owner. The work described in this thesis takes a different approach to configuring components, by using tangible interaction techniques instead of GUI-based interaction.

The e-Gadgets project in Section 2.1.5 also made use of a web application to configure components.

What can be learned from the SpeakEasy project is the importance of describing the interfaces of components, such that they can be combined with other components. These interface descriptions help to enable serendipitous interoperability, and are described in more detail in Chapter 7.

2.1.2 *EventHeap (circa 2000-2005)*

Stanford University's shared event system, called the EventHeap, provides a base set of capabilities that link devices in a room [126]. It allows users to move data and applications between areas, for example redirecting a pointer from one device to another. One of these devices, the DynaWall, is a wall-size touch-sensitive interactive display. Gesture-based interaction facilitates moving information objects from the wall from one side to another, by throwing and shuffling visual objects with different accelerations and sounds.

We explored similar interaction techniques during the development of the Spotlight Navigation device, see Section 4.3.3 and [95].

During the development of their system, they identified the following design guidelines:

- Heterogeneity - Devices must be able to interoperate in spite of heterogeneity in software. Interfaces must be customised to work smoothly on different-sized displays with different input/output modalities.
- Dynamism - A software framework must handle applications and devices joining and leaving, while minimising the impact on other entities in the space.
- Robustness - Users will treat the devices in interactive workspaces as appliances that should not fail in inexplicable ways. Devices must provide for quick recovery.
- Interaction techniques - A long, large wall needs an interaction technique suited to its size and location (such as DynaWall's throwing and shuffling technique).

Tuple spaces were first discussed in Section 1.2.1.

Devices in EventHeap use a *tuple space protocol* to communicate with one another, where particular tuples have meaning to certain parties [37]. This semantic agreement between parties is implemented by the developer, for example a tuple representing a request to scan an image.

The iStuff toolkit [5] was developed within Stanford to explore post-GUI interaction techniques, and makes use of the EventHeap system. The iStuff toolkit allows users to map wireless input devices like buttons, sliders, wands, speakers and microphones to different applications running on devices like the DynaWall.

Patch Panel [6] is a mechanism in the iStuff toolkit that tries to solve incremental integration, the problem of integrating new devices and applications that may not have a priori knowledge of each others existence or function. Patch Panel uses an approach called intermediation, with a decoupled communication model (such as publish/subscribe) for inter-component communication. Patch Panel uses a set of mappings between triggers and output events to enable intermediation. These mappings are defined using the Patch Panel Manager GUI or Finite State Machine (FSM)-based scripting language, and users configure connections using a web-based configuration wizard.

A similar decoupled model was used for the work described in this thesis.

Existing toolkits like iStuff do not provide support for the association of high-level semantics to physical objects [105]. While our approach to sharing information between devices is similar to that of EventHeap and Patch Panel, it differs in in the following ways:

- We use ontologies to describe device capabilities and interaction events
- We use semantic reasoning to improve interoperability between devices
- Tangible interaction is used instead of GUI-based interaction to configure connections

2.1.3 *The XWeb architecture (circa 2001-2003)*

The goal of the XWeb architecture is to allow for *opportunistic assemblies* of interactive resources in order to accomplish a particular task. Olsen et al [89] recognised that both an interactive model for acquiring and using the interaction resources, as well as an underlying infrastructure is needed. In their model, each interaction resource resolves user intent independently, instead of merging inputs from a variety of modalities.

A client-server architecture was used to create the infrastructure, with Extensible Markup Language (XML) objects used to model resources and services. Tasks were defined using a two-part Uniform Resource Locator (URL) in the form `dataReference::viewReference`, where the *view* is an abstract definition of a particular interaction. These views are defined as a tree of *interactors*, where the data and the view of the current task as well as the path of the interactor is used to characterise the current state of the device.

XWeb uses a subscribe mechanism to allow multiple clients to share their information, where the devices themselves are not aware of each other but can still be integrated into the same task. The problem that is addressed is that the different devices can be connected without requiring a lot of configuration effort from the user.

Pierce and Mahaney [91] extended the XWeb approach to opportunistic assemblies with *opportunistic annexing*, which is is the process of temporarily attaching one or more resources, like a speaker or a keyboard, to a device in order to enhance its capabilities. Opportunistic annexing differs from the other approaches in this section in that it extends the existing capabilities of devices, instead of assembling heterogeneous devices into a larger, aggregate device.

Pierce and Mahaney expect that the primary benefit of annexing input resources will be faster input rates. This means that the actual annexing action should be faster than the time required to perform the action. For example, if a user will save 5 seconds by typing a note on a keyboard rather than on a mobile device, annexing the keyboard to the mobile device should take less than 5 seconds.

In our approach we go beyond XML-based descriptions to modelling resources and services, by using ontologies to describe devices, and performing semantic reasoning to discover the different ways these devices can be connected to one another.

2.1.4 *AutoHAN (circa 2001)*

AutoHAN is a networking and software architecture to enable user-programmable specification of interaction between appliances in a home environment [15]. It tries to solve the issue of potential complexity between digital devices that interact with each other, especially if these devices are manufactured by different companies (as it is the user who has to specify how they will interact with one another).

The representations of AutoHAN's abstractions are notational systems, validated by the Cognitive Dimensions framework discussed in more detail in Section 11.2.2.

Blackwell distinguishes between two different abstractions that users have to consider [15]:

- *Abstraction over time*, where an appliance has to do something in the future, for example recording a TV programme
- *Abstraction over a class* of entities, where the user is referring to a set of entities, for example a music playlist

Within the AutoHAN project the *Media Cubes* language was created — a tangible representation of an abstract situation. Each cube has a button for input, and a LED and piezo-electric transducer for feedback. Cubes communicate with the AutoHAN network via infrared ports, and use induction coils on four faces of the cube to detect proximity to other cubes. By holding one face of a cube against an appliance, the cube can be *associated* with some function of that appliance. Each individual cube is regarded by the user as a direct manipulation interface to some appliance function, where many different devices may implement this function. This is in contrast to a remote control, that is dedicated to a single appliance but provides access to many different functions.

The cubes of the AutoHAN project can be viewed as a forerunner to the cubes used with our Interaction Tile (described in Section 3.3.1), except that the AutoHAN cubes represent tasks, whereas the Interaction Tile cubes represent devices.

Each cube has a unique identifier, and each face of the cube can also be identified. This means that a combination of cubes and neighbouring faces can be used as a type of programming language. Cubes may also be associated with virtual devices: software components running somewhere on the network. The user regards these virtual devices to be the same as physical appliances that are placed in the broom cupboard, like a network router or home server.

AutoHAN devices communicate using UPnP Generic Event Notification Architecture (GENA). UPnP control points and services use GENA to implement eventing. GENA is a publish/subscribe system that uses HTTP as transport mechanism. Conceptually, UPnP control points are subscribers, while UPnP services are publishers [62]. GENA defines three new HTTP methods to manage event subscriptions and deliver messages:

- SUBSCRIBE to subscribe to event notifications and renew existing subscriptions
- UNSUBSCRIBE to cancel a subscription
- NOTIFY to send an event notification to a subscriber

AutoHAN entities make subscription requests to receive certain types of events. When such an event occurs, an HTTP NOTIFY request is sent by the AutoHAN server to the subscriber, with additional parameters (such as which button on a control panel was pressed) are encoded in the GENA Notification subtype or in the message body.

Two alternative programming paradigms were considered for the Media Cubes language - an *ontological paradigm* and a *linguistic paradigm*. In the ontological paradigm, tokens represent "natural categories" in the user's mental model. Concepts were identified which have a close correspondence between primitive remote control operations, appliance functions, capabilities and user skills, representing a primitive ontology of home automation. These abstract types were incorporated into four types of cubes:

- An *Event* cube ("on"/"off", "go"/"stop") to represent a change of state, such as a sensor activation (e.g. a doorbell) or automated function (e.g. alarm clock). "Go" and "on" is functionally identical, but labeled separately to help users reason about equivalence between events and processes.
- A *Channel* cube can be used to associate a media channel/stream with a media source, and direct the stream to a media sink.
- An *Index* cube selects content from a channel and can be associated with particular index values, to select content that matches that value.
- An *Aggregate* cube allows the user to refer to abstract collections rather than individual instances.

In the linguistic paradigm, cubes represent words in a language, for example a single face of a cube may be labelled *Clone*. When this face is placed against another cube face and activated, the second face takes on the identity and function of the first. A *List* cube has three active faces: Add Item, Remove Item and Contents.

In our approach we try to improve on the ontological paradigm used in the AutoHAN project, by expanding on how the events and channels/connections are modelled. In addition to the publish/subscribe approach used in AutoHAN, we make use of a blackboard architectural pattern to share information between devices.

2.1.5 *e-Gadgets (circa 2004)*

The e-Gadgets[1] project was a European project within the Disappearing Computer initiative[2]. An architectural style, called Gadgetware Architectural Style (GAS), was developed for devices to communicate with one another. To evaluate GAS, a supporting infrastructure and computationally enhanced artefacts, called e-Gadgets, were created.

Mavromatti et al. [73] developed an approach to allow users to treat these e-Gadgets as reusable components which can be connected to one another. They defined the following requirements for such a system:

- Devices should interoperate via a universal system architecture that accommodates existing communication technologies, e.g. WiFi and Bluetooth.
- Tools and interfaces should allow people to control devices and services. These can either be contained within existing devices or created for a specific purpose.
- Invisible connections will exist between the different physical and virtual devices. Tools must visualise this device structure, make device states visible, explain device functionality and help people to manage the inter-device associations.

The GAS defines a set of concepts and rules in an ontology, a middleware, a methodology and a set of tools that enable people to compose distributed applications using services and devices in an ubiquitous computing environment. At the conceptual level, GAS specifies a *plug-synapse* model, where device capabilities are visualised in the form of *plugs*. Plugs can be associated with one another, creating *synapses* between devices.

Plug descriptions are defined in XML, using a DAML+OIL ontology, and linked to a unique device identifier. DAML+OIL, a combination of the DARPA Agent Markup Language (DAML) and Ontology Inference Layer (OIL) markup languages, has been superseded by Web Ontology Language (OWL).

Synapses and plugs are viewed and modified using an GUI editor. A concept evaluation, using a usability testing approach, was performed with the editor to test the comprehensibility of

1 http://extrovert-gadgets.net/
2 http://www.disappearing-computer.net/

the concepts and the willingness to use such a technology. The Cognitive Dimensions (CD) framework was used to perform a heuristic evaluation. This framework was also used an evaluation of the work described in this thesis, and is discussed in more detail in Section 11.2.2. Results from the evaluations include the following:

- Users will use their experience gained through a trial-and-error process to bridge the gap between their intentions and the feedback gathered through their actions.
- A device can be part of multiple in-home applications at the same time. The effect from interacting with that device is not clear based on physical appearance alone.
- A state change in one device could create a non-visible state change on another device.

They also noted that the possibility to combine the functionality of devices opens up possibilities for emergent behaviour, where the emergence results from how the devices are actually used. The evaluation revealed that this can be confusing, as the user has to guess what interfaces the system allows or does not allow.

In our approach we use an OWL 2 ontology instead of DAML+-OIL, and use a reasoning engine to perform semantic matching of device capabilities. As already mentioned earlier in this section, we tried to move beyond the traditional GUI-based approach to configure the connections between devices. We also used different kinds of feedback and feedforward to make emergent behaviour possibilities clearer for the user.

Apart from the projects and frameworks presented here, we also want to look at the other aspects that are related to the work described in this thesis. The rest of the chapter will introduce ontologies considered to be state-of-the-art in ubiquitous computing, as well as related user interface software architectures, input device modelling methods and semantic models.

2.2 UBICOMP ONTOLOGIES

In this section we will look at the various ubicomp ontologies that have been developed for context-aware computing. The ontologies described later in this thesis builds upon this existing work, but with a stronger focus on interaction-related aspects.

2.2.1 *SOUPA (circa 2004)*

Chen et al. [24] created Standard Ontology for Ubiquitous and Pervasive Applications (SOUPA), a context ontology based on OWL, to support ubiquitous agents in their Context Broker Architecture (CoBrA). The ontology supports describing devices on a very basic level (e.g. typical object properties are `bluetoothMAC` or `modelNumber`), but it has no explicit support for modelling more general device capabilities.

In SOUPA, an agent ontology is used to describe the actors in a system, where actors include both human and software agents or computing entities. A computing entity is characterised by a set of mentalistic notions in the Belief-Desire- Intention (BDI) model, such as knowledge, belief, intention and obligation. The properties of a person agent includes basic profile information, like name, gender, and age, as well as contact information, which includes e-mail, phone number, mailing address etc. SOUPA references several domain ontologies to achieve this, for example Friend-Of-A-Friend (FOAF)[3], one of the most well-known ontologies, used to describe people, their activities and relations to people and objects. SOUPA uses FOAF to express and reason about a person's contact profile and social connections with other people.

SOUPA covers contexts in the office/campus environment, but it has no explicit support for modelling general contexts in heterogeneous environments. We now look at the BDI model, and the MoGATU BDI ontology used in SOUPA, in more detail.

In personal communication with the author of the MoGATU ontology, he mentioned that MoGATU is not an acronym, but the name of his PhD project.

2.2.2 *BDI and the MoGATU ontology*

The BDI model is a philosophical model of human practical reasoning originally developed by Michael Bratman [18], with a number of successful implementations and applications in the agent research community [19, 49]. It could be argued that the BDI model is somewhat dated, as the principles of the architecture were established in the mid-1980s and have remained essentially unchanged since then [48].

In a smart environment, we wish to infer a user's intention based on his/her context and interaction with the environment. In BDI theory, a *desire* is the motivational state of an agent, with a *goal* having the added restriction that multiple active desires

3 http://http://www.foaf-project.org/

must be consistent (e.g. concurrent desires of "going to a party" and "staying at home" is not possible). A user's *intention* is a desire to which the user has committed. *Plans* are a sequence of actions to reach a specific goal. We can therefore infer intention based on an action, or sequence of actions. When an agent commits to a specific plan with subgoals based on a *belief*, or informational state of the agent, it needs the capability to reconsider these subgoals at appropriate times when the beliefs change.

When the goals, plans, desires, and beliefs of different agents are explicitly represented in an ontology, this information allows them to share a common understanding of their "mental" states, helping them to cooperate and collaborate. If we are able to represent the human user's mental states in the ontology, it may help software agents to reason about the specific needs of the users in a pervasive environment.

MoGATU BDI, an ontology developed by the same research group that developed SOUPA at the University of Maryland [129], describes an abstract semantic model for representing and computing over a user's or an agent's profile in terms of their prioritised and temporally ordered actions, beliefs, desires, intentions and goals. SOUPA uses this model to help independent agents to share a common understanding of their "mental" states, so that they can cooperate and collaborate. The agents also help to reason about the intentions, goals, and desires of the human users of a system.

2.2.3 *Gaia (circa 2004-2007)*

Ranganathan et al [94] developed an uncertainty model based on a predicate representation of contexts and associated confidence values. They incorporated this model into Gaia, a distributed middleware system for pervasive computing. Contexts are represented as predicates, following the convention that the predicate's name is the type of context being described (such as location, temperature, or time). This gives a simple, uniform representation for different kinds of contexts. Some contexts (such as `office`) are certain, whereas others (such as `location` and `activity`) might be uncertain. Uncertainty is modelled by attaching a confidence value between 0 and 1 to predicates. The context model is represented using DAML+OIL.

While Gaia's focus was on modelling the uncertainty of context in ubiquitous computing environments, our focus is more on modelling the connections between devices in such an environment, as well as the interaction events that occur when people operate these devices.

2.2.4 *CAMUS (circa 2004-2005)*

Ngo et al. [77] developed the Context-Aware Middleware for Ubiquitous computing Systems (CAMUS) ontology in OWL to support context awareness in ubiquitous environments. Their device ontology is based on the Foundation for Intelligent Physical Agents (FIPA) device ontology specification[4], with every `Device` having the properties of `hasHWProfile`, `hasOwner`, `hasService` and `hasProductInfo`. Devices are further classified into `AudioDevice`, `MemoryDevice`, `DisplayDevice`, or `NetworkDevice`. For audio, the `hasParameter` property has the `AudioParameter` class as range, with subclasses like `ACDCParameter`, `Intensity` and `HarmonicityRatio`.

One of the major goals of context-aware computing is to provide services that are appropriate for a person at a particular place, time, situation etc. In CAMUS, context entities and contextual information are described in the ontology as well [77]. For the entities related to agents, there is a top level concept called `Agent`. It has been further subclassed into `SoftwareAgent`, `Person`, `Organization`, and `Group`. Each `Agent` has a property `hasProfile` associated with it, whose range is `AgentProfile`. An `Agent` is also related through the `isActorOf` relationship to an `Activity`.

There are some conceptual modelling issues with CAMUS, for example having organisations and groups being direct subclasses of the `Agent` class. An issue that is not addressed by CAMUS or the other ontologies is how to model user interaction, which is the focus of the next section. We consider a number of user interface software architectures that can be used to model user interaction in a smart environment.

2.3 USER INTERFACE SOFTWARE ARCHITECTURES

A user interface software architecture, also known as a User Interface Management System (UIMS), creates a separation of concerns between the user interface and the implementation of a software application or system. Of these, the Model-View-

4 http://www.fipa.org/specs/fipa00091/SI00091E.html

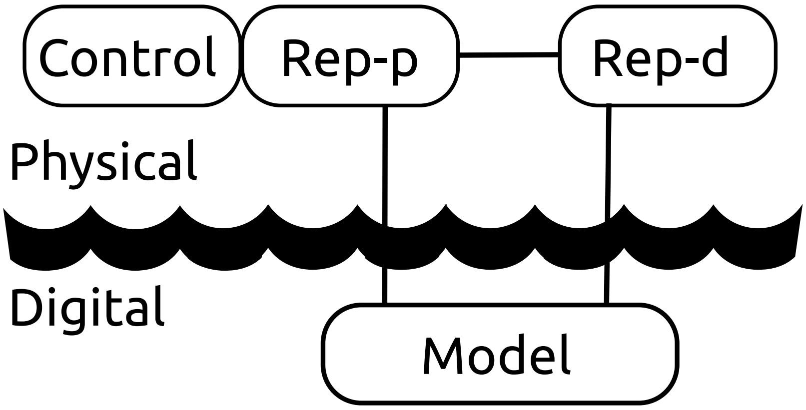


Figure 3: The MCRpd model of Ullmer & Ishii for TUIs

Controller (MVC) model is currently the most used, and inspired Ullmer & Ishii's [112] Model-Control-RepP-RepD (MCRpd) interaction model for tangible user interfaces. Other UI software architectures include the Arch/Slinky model [10] and the Adapter, System, User, Real object (ASUR) interaction model.

2.3.1 *Ullmer & Ishii's MCRpd model*

TUIs in general attempt to use the physical appearance of an object to communicate its virtual affordances [11]. A user working with a GUI only manipulates virtual objects, whereas TUIs allow the user to manipulate both physical and virtual objects, which coexist and share information with each other [105]. In a TUI, the behaviour of a physical object is determined by the object's interactions with other physical and virtual objects - this is also the case in a smart environment.

Ullmer and Ishii [112] extended the traditional MVC model for TUIs, as shown in Figure 3. They distinguish between the physical and digital domains by placing the physical domain above the waterline, and the digital domain below the waterline. The model element is carried over from the MVC model and represents the intangible digital information. The control element is also carried over from the MVC model, while the view element is split into two subcomponents:

- Physical representations (Rep-p) – represents the physically embodied elements of tangible interfaces
- Digital representations (Rep-d) – represents the computationally mediated components of tangible interfaces with-

out embodied form, for example video and audio, that can be observed in the physical world

In a tangible interface, the physical representations (Rep-p) are computationally coupled to the underlying digital information (model), as well as perceptually coupled to the computationally mediated digital representations (Rep-d). The interaction model introduced in Section 6.1 was inspired by the MCRpd model.

2.3.1.1 *Arch/Slinky model*

Bass et al [10] contend that no single software architecture will satisfy all the design goals of an interactive system. With the Arch/Slinky model the buffering of a system from changes in technology was selected as the most important criterion. Here are some of the other design criteria they defined, which we consider to be especially important to ubiquitous computing systems:

- target system performance (e.g. size and speed)
- buffering from changes in application domain and hardware platform
- conceptual simplicity
- target system extensibility
- compatibility with other systems

They define an *application* to be the total system that is developed for its end users, while the *application domain* is the field of interest of, or reason for, the application. They also extended the definition of UIMS to a User Interface Runtime System (UIRS) - the runtime environment of an interactive application.

The Arch model creates a bridge between the physical interaction device and the application domain. The following 5 components are defined:

- Interaction Toolkit Component - implements the physical interaction with the user (also called physical level)
- Presentation Component - provides a set of implementation-independent objects, e.g. a “selector” object can be implemented by both radio buttons or a drop-down menu in a GUI (also called lexical level)

- Dialogue Component - does task-level sequencing and maps between domain-specific and UI-specific formalisms (also called dialogue level)
- Domain Adaptor Component - triggers domain-initiated tasks, organises domain data, detects and reports semantic errors (also called functional core adapter)
- Domain-specific Component - controls, manipulates and retrieves domain data

The separation of functionality into the different components was done to minimise the effects of changes in technology. The Slinky meta-model is a generalisation of the Arch model, providing a set of Arch models with different weights assigned to each component.

The difference between the Arch/Slinky model and our interaction model is that the Arch/Slinky model relates a single physical interaction device with a software application, while our interaction model relates smart objects with one another through semantic connections.

2.3.1.2 *The ASUR interaction model*

ASUR is a notation-based model to describe user-system interaction in mixed interactive systems [36] at design-time. It describes the physical and digital entities that make up a mixed system. ASUR uses directed relationships to express physical and digital information flows, as well as the associations between components.

Both components and relationships may have characteristics. For components, this includes the location where the information is perceived (e.g. top of table) and action/sense required from the user (e.g. sight, touch or physical action). For relationships, characteristics include the dimensionality of the information (e.g. 2D or 3D) and the type of language used (e.g. text or graphics).

A sequence of such entities and their relationships in an interaction forms an *interaction path*. The interaction exchange or action between elements in the path is conducted via one or more *interaction channels* along which information or action is communicated. An interaction channel may be described in terms of its properties, either physical or digital depending on the channel, e.g. a digital channel may be described in terms of bandwidth, uptime and the nature of the connection. Adaptors are used transform information from the physical environment to the digital world and vice versa. An accelerometer for example may be modelled as a separate device, but if integrated into smart phones it can be abstracted away as part of an interaction path.

Interaction carriers are mediating entities that are necessary for information communication. Passive carriers can carry and store part of the information communicated along an interaction path, e.g. a tangible object left in a particular position. Active carriers are transmitters of non-persistent information along the interaction path, e.g. a stylus used to transmit a precise position on a touch screen. Contextual entities are physical entities involved in an interaction (e.g. a table), and are also considered mediating entities.

The intended user model refers to what the user should know about the interaction in order to carry it out successfully. It may refer to one atomic interaction path (e.g. a channel, source and destination), or it may refer to more complex paths.

An interaction group refers to a set of entities and channels that together have properties that are relevant to a particular design issue. Some of these groups will be applicable to any design, while others will depend on the task and context:

- Entities and channels may be *grouped for feedback*, to identify an interaction flow that links the response of the system to the actions of the user.

- User interface elements may be linked to application concepts in order to express a semantic association. The goal is to help the user to cognitively unify elements of the group (helping to establish the intended user model).

- Sets of input (e.g. speech input for gesture input - "put that there") that must be combined to perform a certain task, may be grouped for multimodal interaction.

- A grouping may be used to assert that a set of services must reside on the same machine or be distributed over multiple devices.

- A grouping of paths may show information flows among or between multiple users.

An advantage of the ASUR interaction model is that it combines both the physical and digital dimensions of user-system

INTERACTION TASK	LOGICAL DEVICE	PHYSICAL DEVICE
Position	Locator	Tablet
Select	Choice	Touch Panel
	Pick	Trackball/Mouse
Path	Stroke	Joystick
Quantify	Valuator	Dials
Text entry	String	Keyboard
Orient		

Table 1: Interaction tasks mapped to logical and physical interaction devices

interaction. Later in Chapter 6 we will see how our interaction model also combines both these dimensions.

In this section we looked at how user interface software architectures can be used to model interactive systems. In the next section we look at how input devices can be modelled to describe the different interaction tasks that can be performed in a smart environment.

2.4 MODELLING INPUT DEVICES

2.4.1 *Foley's taxonomy and its extensions*

Foley [42] describes a taxonomy of input devices that are structured according to the graphic subtasks they can perform: position, orientation, select, path, quantify and text entry. He defined these subtasks as six Basic Interaction Tasks (BITs). A BIT is the smallest unit of information entered by a user that is meaningful in the context of the application. He noted that there are far too many interaction techniques to give an exhaustive list, and that it is impossible to anticipate which new techniques may be created. In Table 1 we map them to possible logical and physical interaction devices. The six types of logical devices were also defined by Foley in [43].

Some characteristics of the physical interaction devices are not shown in the table. The positioning of tablets and touch panels are *absolute*, while that of trackballs, joysticks and mice are *relative*. A touch panel is considered *direct*, as the user directly points at the screen, while a tablet is *indirect*. Joysticks, tablets and mice are *continuous*, while a keyboard is *discrete*. Dials can either be *bounded* or *unbounded*.

The *positioning* interaction task involves specifying an (x,y) or (x,y,z) position. Characteristics of this task include different coordinate systems, resolution and spatial feedback. The *select* interaction task involves choosing an element from a choice set, while the *text* interaction task entails entering character strings to which the system does not assign specific meaning. The *quantify* interaction task involves specifying a numeric value between some minimum and maximum value. The *path* interaction task consists of specifying a number of positions over a specific time or distance interval. The *orient* interaction task is also called *rotate*, but is not often used [28].

Card et al [22] argued that the Foley taxonomy has not tried to define a notion of completeness, and is thus not generic enough. They pointed out that single devices appear many times in the levels of the tree, which makes it difficult to understand the similarities among devices. MacKinlay, Card and Robertson [72] extended Buxton's work to propose additional physical properties that underly most devices. They follow mappings from the raw physical transducers of an input device into the semantics of the application.

Dix et al [32] noted that Card et al's analysis is not only relevant to GUIs, as they used a running example of a radio with knobs and dials. Their work not only abstracts devices into classes, but also takes into account that rotating a dial is different from moving a slider, i.e. the physical nature of the interaction is also important.

Ballagas et al [7] surveyed interaction techniques that use mobile phones as input devices to ubiquitous computing environments, and used Foley's six interaction tasks as a framework for their analysis. In their work on iStuff [5] they state that the set of interactions tasks are only sufficient for describing graphical user interfaces, not physical user interfaces, or user interfaces in general. The same paper notes that Buxton's taxonomy, and the extension by MacKinlay, Card and Robertson, is too narrow for ubiquitous computing environments, as it does not classify devices with different modalities and only describes input devices. They extended the taxonomy further to describe attributes like direction and modality. The direction attribute is used to indicate whether a device provides input, output or both. The modality attribute describes different visual, auditory haptic and manual modalities for input and output. Additional attributes they identified include directionality/scope (where a device is targeted to one, many, or all the users in a room) and mount time (the effort necessary to use an interaction device).

Based on the work of Foley, Card and others in this section, we defined a concept called the *interaction primitive*, described in more detail in Section 4.2.2 and 6.2.2. Interaction primitives can be used as a way to describe the user interaction capabilities of smart objects in ubiquitous computing environments.

While these techniques are well suited to modelling input devices, we still need a way to describe the semantics of interaction feedback and the different types of interactions that can occur. That is the focus of the next section on semantic models.

2.5 SEMANTIC MODELS

2.5.1 *The Frogger framework*

The Frogger framework, as was introduced by Wensveen [123], describes user interaction in terms of the information a user perceives (like feedback and feedforward), and the nature of this information. It distinguishes between inherent, augmented and functional information. These types of information can serve as couplings between user actions and the systems' functions in time, location, direction, modality, dynamics and expression. Although the framework was designed to describe the interaction with electronic devices and their interfaces, many of the concepts in the framework are applicable to interactions with systems of devices as well.

When a user performs an action and the device responds with information that is directly related to the function of that product (lighting switching on when a light switch is operated), we speak of *functional feedback*. When a device has more than one functionality, functional feedback should be viewed with respect to the users' intentions and goals when performing the action. If there is no direct link between a user's action and the direct function of the product, or when there is a delay, *augmented feedback* (also known as indicators [109]) can be considered to confirm a user's action. This feedback is usually presented in the form of lights, sounds or labels. *Inherent feedback* is directly coupled (inherently) to the action itself, like the feeling of displacement, or the sound of a button that is pressed.

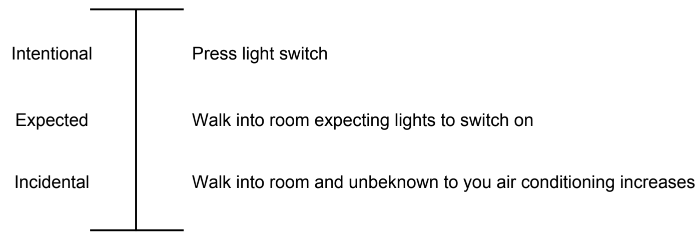


Figure 4: The continuum of intentionality

Affordances were discussed in Section 1.2.3.

While feedback is information that occurs after or during the interaction, feedforward is the information provided to the user before any action has taken place. *Inherent feedforward* communicates what kind of action is possible, and how one is able to carry out this action. Inherent feedforward is the same as the concept of affordances, revealing the action possibilities of the product or its controls [123]. When an additional source of information communicates what kind of action is possible it is considered *augmented feedforward*. *Functional feedforward* communicates the more general purpose of a product.

The concepts described in the Frogger framework are used to implement feedback and feedforward in the work described in this thesis, specifically in Section 5.4.1 and Section 6.6.

2.5.2 *Models of intentionality*

At the semantic level, we are interested in the meaning of the action. A gesture may mean nothing, until it encounters for instance a light switch [16]. In traditional software applications, a user is expected to have a clear intention of what he/she wants to achieve, with purposeful and direct actions. In ubiquitous computing scenarios, the interactions are less explicit. Input can be implicit, sensor-based and "calm", and output is ambient and non-intrusive. With *incidental interactions* [31], a user performs an action for some purpose (say opening a door to enter a room), the system senses this and incidentally uses it for some purpose of which the user is unaware (e.g. adjust the room temperature), affecting the user's future interaction with the system.

The *continuum of intentionality* in Figure 4 has normal, intentional interactions at the one end of the spectrum (e.g. pressing a light switch), expected interactions in the middle (e.g. walking into a room expecting the lights to go on), and incidental interactions at the other end. As users become more aware of the interactions happening around them, they move through the continuum toward more purposeful interaction. For example, with *comprehension* an incidental interaction (lights turning on when you enter the car) turns into an expected interaction. With *co-option*, an expected interaction turns into an intended interaction (e.g. deliberately opening and closing the car door to turn on the light).

Incidental interactions do not fit existing interaction models based on the conventional intentional cycle, like Norman's Action Cycle Diagram [84]. The purpose of the user's activity is distinct to the intended outcomes of the system. Feedback may be unobtrusive (and not noticed), or delayed (like the temperature slowly changing). There are two tasks that are occurring: The user's purposeful activity, and the task that the incidental interaction is attempting to support/achieve.

We try to improve this comprehension by actively involving users in configuring the relationships between the smart objects in their environment. Bellotti et al. [11] posed five questions to the designers of ubiquitous computing technologies:

- Address: How do I address one (or more) of many possible devices?
- Attention: How do I know the system is ready and attending to my actions?
- Action: How do I effect a meaningful action, control its extent and possibly specify a target or targets for my action?
- Alignment: How do I know the system is doing, or has done, the right thing?
- Accident: How do I avoid mistakes?

When users are able to explore and manipulate the relationships between the smart objects, it becomes easier for them to begin to comprehend how things work, or can potentially work together. They can project their experiences with a part of a smart environment to see what may potentially work for other parts of the environment as well. By allowing users to configure their smart environment themselves, they are in control of deciding how the environment responds to their actions. By making use of the feedback mechanisms introduced in Section 2.5.1 we can indicate that the system is ready and attending to a user's actions. These mechanisms can also be used to make the action possibilities of the system more visible and help the user avoid mistakes.

2.6 OUTLOOK

As Nielsen [78] noted, the purpose of a model is to improve the usability of software. He noted that some people will consider a model a useful abstraction, while others will prefer other models, similar to how everybody has their own favourite programming language.

We build further on many of the concepts and proposals reviewed in this chapter. In particular, we focus on configuring the connections between the devices in Chapters 3 to 5 and Chapter 6, while serendipitous interoperability and sharing information between devices form the cornerstones of Chapter 7 and Chapter 8.

Part II

DESIGN ITERATIONS AND CONSTRUCTING A THEORY

An iterative development process was followed for the work described in this thesis. In this part of the thesis, we describe the three design iterations. Extracting from the lessons learned during the three design iterations, a theory of semantic connections is introduced.

3 DESIGN ITERATION I

I think the only way forward is going from applying algorithms to individual transactions, to first placing information in context — pixels to pictures — and only applying algorithms after one sees how the transaction relates to the other data. It's the only way that I can see that it's going to close this sense-making gap.

— Jeff Jonas [127], data scientist

Parts of this chapter appear in [80] and [117].

An iterative development process was followed for the work described in this thesis. In the following chapters three iterations, each consisting of a requirements and planning phase, analysis and design phase, implementation phase and evaluation phase, is described in more detail. Iterative processes are essential to modern-day software and hardware development methodologies, exemplified by the various agile development frameworks [68].

3.1 REQUIREMENTS

The goal of the first design iteration was to see if using a tangible interface to establish connections between devices is a viable alternative to the usual GUI-based solutions. Additionally, different approaches to modelling user interaction, device capabilities and connections were explored.

Scenarios are commonly used in software engineering and interaction design to help discover and analyse requirements. The following scenario was presented at the start of the project to guide the design process:

Mark is a 12-year-old boy and he is at home receiving his friend Dries from school. Dries arrives with a portable music player loaded with his favourite songs. He wants to play some recent collections for Mark. Mark's home is equipped with a sophisticated surround sound system, and they have recently installed an ambient lighting system that is connected to the sound system and renders the mood of the music by dynamic colour lighting in the room. They decide to use both to enjoy the music. Dries starts streaming his music to the environment.

An object (or several objects) shows possible input and output ports for streaming music in the environment. By interaction with the object/objects, Mark connects the output from Dries' music stream to the input of the sound system. Now the room is full with Dries' music and the colourful lighting effects. Mark's mom, Sofia, now comes back from work. She starts preparing dinner for the family. Mark and Dries don't want to bother her with their loud music. They again use the object(s) to re-arrange the music stream. Now the music is streamed to Mark's portable music player while playing back at Dries'. It is also connected to the ambient lighting system directly, bypassing the sound system. They both are enjoying the same music using their own favourite earphones, and the colourful lighting effects, but without loud music in the environment.

The object(s) shows the connection possibilities with a high level of semantic abstraction, hiding the complexity of wired or wireless networks. By interacting with the object(s), semantic connections can be built, redirected, cut or bypassed.

The first takeaway from this scenario is that the focus is on the connections between the devices, instead of on the devices themselves. This brings us to the first design decision: *Semantic connections* are introduced as a means to indicate users' intentions concerning the information exchange between smart objects in a smart environment.

SEMANTIC CONNECTION A semantic connection is a relationship between two entities in a smart environment for which we focus on the semantics—or meaning—of the connections between these entities.

The term semantic connections is used to refer to meaningful connections and relationships between entities in a smart environment. These connections are both real "physical" connections (e.g. wired or wireless connections that exist in the real world) and "mental" conceptual connections that seem to be there from the user's perspective. The context of the connections, for example the objects that they connect, provide meaning to the connections. The term "semantics" refers to the meaningfulness of the connections. The type of connection, which often has the emphasis now (e.g. WiFi, Bluetooth or USB) is not considered to be the most relevant, but what the connection can do for someone — its functionality — even more.

The following requirements were defined during this phase:

- Semantic connections exist in both the physical and the digital world. We need ways to visualise these invisible connections and to control them.

- Devices need to be able to share their capabilities and content with the other devices in their environment.

A number of different approaches to visualising and controlling semantic connections were explored in the first iteration, and these are described in Section 3.3. We also need a way to model the devices, their capabilities and the connections themselves. This is the subject of the next section.

3.2 ONTOLOGY DESIGN

OWL 2, the ontology language used to build ontologies for the Semantic Web, was used to create the ontologies in this thesis. OWL 2 has been a W3C Recommendation since October 2009, and adds new capabilities like property chains to the original OWL standard.

Ontologies and ontology engineering are described in more detail in Section 9.

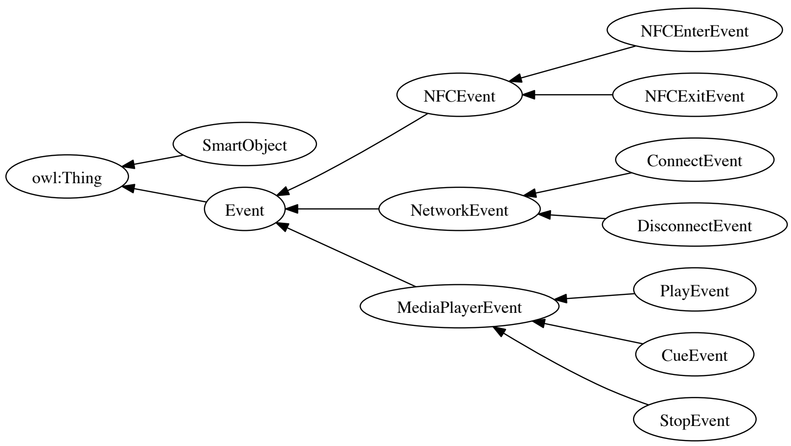


Figure 5: Ontology indicating subclass relationships

A first attempt at modelling the various entities in an ontology is shown in Figure 5. A bottom-up approach to modelling was used, where we attempted to model each entity using the least number of statements. These entities were later aligned with foundational ontologies – the approach that was followed is discussed further in Chapter 9. Each entity is modelled as an `owl:Class`, where all classes are subclassed from the root class, owl:Thing[1]. Each edge in the graph above is an rdf:type relationship, and the direction of the arrow indicates the direction of the subclass relationship.

During the initial development stages, we realised that the most promising way of describing low-level interactions seemed to be to describe them in terms of *interaction events*, that are defined as follows:

Interaction events are discussed in more detail in Chapter 8.

INTERACTION EVENT An interaction event is defined as an event that occurs at a certain time instant and was generated by a specific smart object. It reports either the intent of a user's action directly, or a perceivable change in a smart object's state.

An interaction event in the smart space consists of an event ID, timestamp and other related information, like the smart object that generated that event. For the scenario, three types of interaction events were defined:

- Network events: A ConnectEvent indicates that a device is entering the smart space, while a DisconnectEvent means that the device is exiting the smart space.
- Near Field Communication (NFC) events: An NFCEnterEvent signifies that an NFC tag has entered the RFID field, and a NFCExitEvent is generated when it leaves the field.
- Media player events: When the user presses the Play button on the media player, a PlayEvent is generated. When the music is stopped, or at the end of the song, a StopEvent is generated. Pressing the Forward button forwards the song by 5 seconds. This time period is attached to a CueEvent using an atTime relationship.

All OWL code listings in this thesis are written using Turtle[2] syntax. Turtle is a human-friendly alternative to XML based syntaxes.

The following properties were defined:

```
:connectedTo
    a owl:ObjectProperty;
    a owl:IrreflexiveProperty;
    a owl:SymmetricProperty ;
    rdfs:domain :SmartObject ;
    rdfs:range :SmartObject .
```

1 The owl: prefix is used to denote the OWL 2 Namespace Document located at http://www.w3.org/2002/07/owl.

```
:atTime
    a owl:DatatypeProperty ;
    rdfs:comment "At a specific time (in milliseconds)";
    rdfs:range xsd:integer .

:generatedBy
    a owl:ObjectProperty ;
    rdfs:domain :Event ;
    rdfs:range :SmartObject .

:hasPosition
    a owl:DatatypeProperty ;
    rdfs:range xsd:integer .

:hasRFIDTag
    a owl:DatatypeProperty ;
    rdfs:range xsd:string .

:inXSDDateTime
    a owl:DatatypeProperty ;
    rdfs:range xsd:dateTime .
```

The `connectedTo` object property is both *symmetric* and *irreflexive*. Irreflexive properties are a new feature in OWL 2. A symmetric property is its own inverse, which means that if we indicate a `connectedTo` relationship from device A to device B, device B will also have a `connectedTo` relationship to device A. Another way to think of symmetric properties is that they are bidirectional relationships.

An irreflexive property is a property that never relates an individual to itself [54]. This allows us to restrict our model by not allowing a `connectedTo` relationship from a device to itself.

An example with individuals, also called instances, that make use of the ontology is shown in Figure 6. In the figure, classes are denoted with ellipses, individuals with boxes and datatypes as plain text. Class membership is denoted with dotted lines and relationships are denoted with solid lines. It shows a Nokia N900 and N95 smartphone instantiated as `SmartObjects` with their associated Radio Frequency Identification (RFID) tags.

Why an RFID tag? In Section 6.2.1 we argue that each smart object must be uniquely identifiable in the physical world by digital devices.

An instantiated `NFCExitEvent`, called `event-1cecdba5`, is also shown. When an event is generated a Universally unique identifier (UUID) is assigned to it, to enable the event to be uniquely identified in the smart space. It is also associated with a smart object using the `generatedBy` property. The `hasPosition` relationship provides additional metadata required by the interaction tile, which is described in the next section.

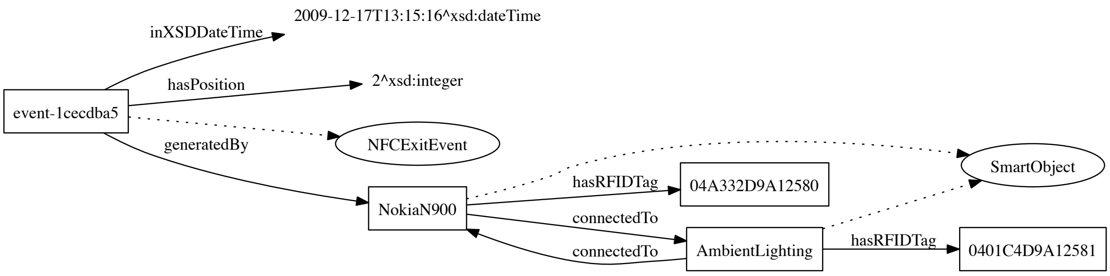


Figure 6: Individuals that were instantiated based on the ontology

SPARQL Protocol and RDF Query Language (SPARQL)[4] form the query language for the Semantic Web. Along with OWL, it is one of the core technologies of the Semantic Web, having been a W3C Recommendation since January 2008. SPARQL queries are based on the idea of graph pattern matching [104], where data that is returned from the query is set to match the pattern.

We also make use of SPARQL to define rules, which is described in Section 9.4.

To determine which other smart objects a specific device, for example a mobile phone, is connected to, a simple SPARQL query suffices:

```
SELECT DISTINCT ?object WHERE{
:phone1 :connectedTo ?object .
}
```

A *triple store* is used to store both the instances and the ontology. A triple store is a purpose-built database for storing and retrieving triples, in the format subject-predicate-object. In the above example *phone1* would be the subject, *connectedTo* the predicate and *?object* the object. There are a number of commercial and open-source triple store implementations. The Jena[5] framework is a Java API that enables access to many triple store implementations, supports SPARQL and also has its own persistent triple store. It was used in this first implementation and was also later adopted by the SOFIA project.

An advantage of using SPARQL and a triple store is that it is easy to add additional constraints and/or specifics to the query, compared to a traditional Structured Query Language

4 http://www.w3.org/TR/rdf-sparql-query/
5 http://jena.apache.org/

(SQL) database, where unions between columns and tables can get quite complicated very quickly.

To get the last event that was generated by a specific device, the SPARQL query is a little bit more complex, but still surprisingly manageable:

```
SELECT ?event ?eventType WHERE{
:deviceID :hasRFIDTag ?tag .
?event :hasRFIDTag ?tag .
?event a ?eventType .
?event :inXSDDateTime ?time .
FILTER (?eventType = :NFCEnterEvent  || ?eventType = :NFCExitEvent)
}
ORDER BY DESC(?time)
```

Semantic data focuses on the relationships between entities, making semantic models property-oriented. Entities are members of a class based on their properties/predicates.

How do we model the semantic connections between devices? Since semantic modelling is property-oriented instead of object-oriented[103], we started by focusing on the possible predicates that can be used to describe connections. We need a way to model whether a connection is possible — this can be done with a `canConnectTo` property. We also need to know if a device is currently connected to another device — `connectedTo`. Then we need a way to model the capabilities that each device provides. In this first iteration, we defined two properties called `consumes` and `provides`. They are used as follows:

```
NokiaN900 provides AudioCapability .
NokiaN95 consumes AudioCapability .
```

During the later design iterations we decided to model capabilities as functionalities of a device instead, and make the name of the property clearer to indicate whether it is a functionality of a source or a sink. The property `provides` was changed to `functionalitySource`, and `consumes` was changed to `functionalitySink`. These early properties are mentioned here for the sake of completeness, and to show how aspects of the ontology have changed between iterations.

3.3 DEVICE DESIGN

Based on the scenario, a number of smart objects had to be constructed or repurposed, and the necessary software had to be developed.

To explore the different possibilities of visualising and manipulating connections between devices, a number of different prototypes were constructed. The first of these is called the *interaction tile*.

3.3.1 *Interaction Tile*

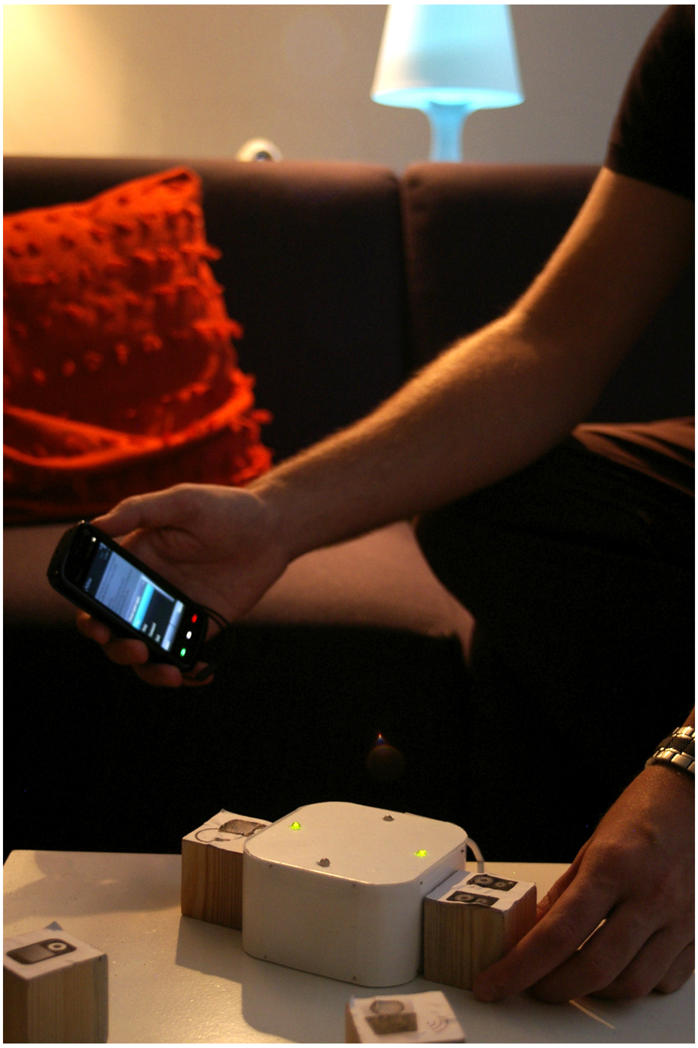


Figure 7: The interaction tile and mobile phone

The interaction tile, shown in Figure 7, was inspired by Kalanithi and Merrill's "Siftables" cubes [74]. It was designed to explore and manipulate connections through direct manipulation – by making simple spatial arrangements. Each device in the smart environment is represented by a cube containing an RFID tag and a small magnet, with an icon on the top of the cube to signify the device being represented. When a cube is placed next to one of the four sides of the tile, an LED on the tile lights up to indicate that it has been recognised. When a second cube is placed next to the tile, the following LED visualisations are used:

- Pulsating green light - a connection is possible
- Constant green light - a connection exists
- Red light - no connection is possible

The interaction tile visualises the various connections by allowing a user to explore which objects are currently connected, and what connections are possible. By means of putting a cube representing a device close to one of the four sides of the tile, a user can check if there is a connection, and if not, whether a connection is possible. By shaking the tile it is possible to create a connection between two devices, or where there is an existing connection, to break the connection. The interaction tile consists of the following components:

- Arduino Duemilanove with Atmel ATMega328 microcontroller
- ACR122/Touchatag 13.56MHz RFID reader
- RF Solutions ANT-1356M 13.56MHz RFID Antenna Coil
- Multi-colour LEDs
- Accelerometer
- Vibration motor
- Piezoelectric speaker
- Magnetic switches

The Arduino communicates with a PC via a serial interface over USB, while the RFID reader uses Personal Computer/Smart Card (PC/SC) drivers over USB. The accelerometer is used to measure when the user is shaking the tile, while the vibration motor and speaker provide haptic and auditory feedback. The magnetic switches are used to determine which side of the tile a cube has been placed. The final laser-cut version of the interaction tile prototype is shown in Figure 8.

The RFID reader component has been tested under Windows, Linux and Mac OS X.

Two alternative designs are presented in Van der Vlist's thesis [116]. A more detailed discussion of the interaction tile and how its design is informed by product semantics is available in [117].

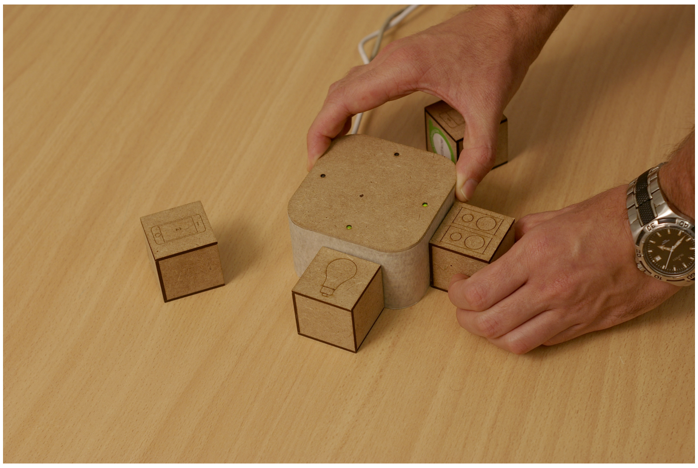


Figure 8: A laser-cut version of the interaction tile prototype

3.3.2 *Lamp*

To create the ambient lighting system, we replaced the internals of a table lamp with an RGB LED array and an Arduino[6]. A Bluetooth module was connected to the Arduino to facilitate communication with a computer, the final result of which can be seen in Figure 9.

The Ikea Lampan lamp that was used for the prototype currently retails for around €3.

The coloured lighting can be changed by sending three RGB values (in the range 0-255) to the lamp via the serial-over-Bluetooth interface.

3.3.3 *Mobile phones*

For the first iteration, a Nokia N95 and Nokia 5800 XpressMusic phone (shown in Figure 10) were used. The two phones use the Symbian S60 operating system, and Python for S60 was used to write software for the mobile phones.

Python for S60 is Nokia's port of the Python programming language for Symbian devices.

3.3.4 *RFID reader used in interaction tile*

Most of the RFID readers and tags targeted at the amateur and hobbyist markets, like the PhidgetRFID and Innovations ID-12 modules, operate in the 125KHz range. While they are relatively cheap and readily available, the 125KHz readers cannot read multiple tags within range of the reader at the same time. For this a 13.56MHz reader is required. The most widely used RFID tags at the moment, the MiFare range owned by NXP, op-

6 http://www.arkadian.eu/pages/219/arduino-controlled-ikea-lamp

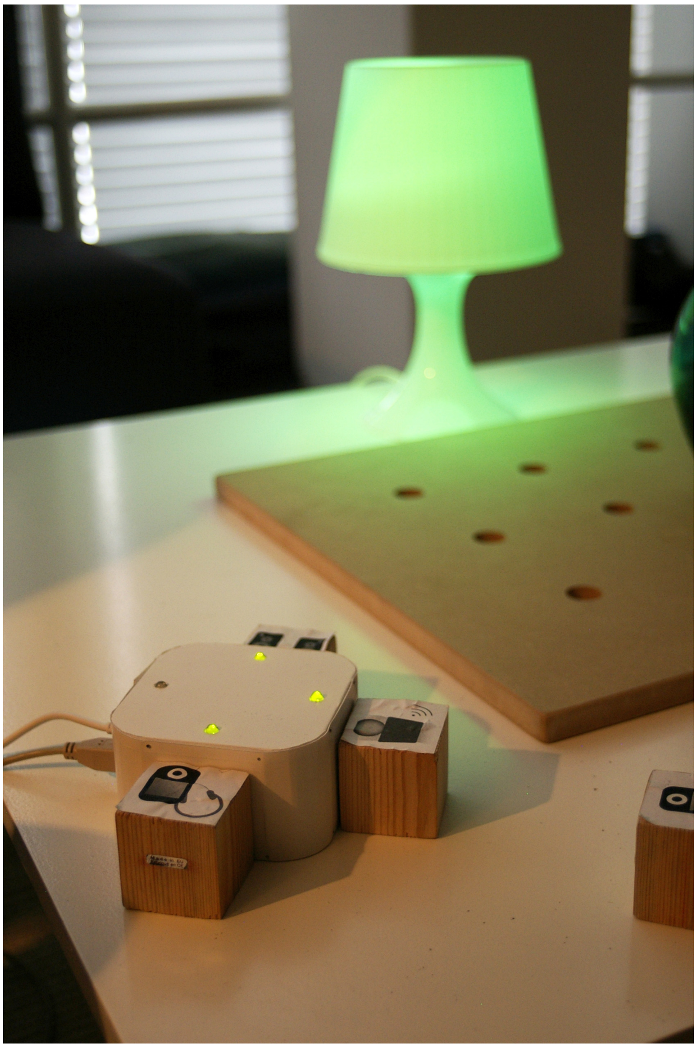


Figure 9: The interaction and cubes, with the lamp in the background

erate at 13.56MHz. These tags are used in most public transport payment systems, including the London Oyster Card and the Dutch OV-Chipkaart system.

A relatively cheap 13.56MHz RFID reader system, the ACR122, is developed by Hong Kong-based Advanced Card Systems (ACS). It uses the NXP PN532 chip to read RFID tags. The reader has an onboard PCB antenna – to extend the range of the unit we removed two capacitors on the PCB and soldered in an external ANT-1356M coil antenna from RF Solutions.

A rebranded version of the ACR122, called the Touchatag[8], is currently sold with 10 tags for around €30.

3.4 IMPLEMENTATION

Following the design and development of the ontology and required devices, a demonstrator that implements the scenario

Figure 10: The Nokia 5800 XpressMusic mobile phone with the lamp and some cubes

was created. A visual overview of the demonstrator can be seen in Figure 11. A video of the scenario is available[9].

Each device in the demonstrator is represented by a KP software module. KPs communicate via the SIB, as shown in Figure 12. As discussed in Section 1.2.1, the SIB acts as an information broker, distributing messages between devices. This was an early design decision to reduce coupling, by minimising direct communication between devices, with all messages relayed via the SIB. This philosophy of having a blackboard architectural model, where devices can write to and read from, was followed through all subsequent design iterations. The technical implementation of the various KPs are described in the following subsections.

The system architecture model is described in more detail in Chapter 10.

3.4.1 *Interaction Tile KP*

The interaction tile KP was written in Python and tested on Ubuntu Linux 10.04. On startup, the KP connects to the Arduino inside the interaction tile via the serial-over-USB interface. It establishes a connection with the SIB, after which it connects to the RFID reader inside the tile.

The open-source rfidiot.org library was used to communicate with the RFID reader.

The KP then enters an event loop, waiting until a cube is placed next to the tile. When this happens, the Arduino sends the position of the cube next to the tile to the KP via the serial interface. The RFID tag is read, and a `NFCEnterEvent` is generated. After the RFID tag is read it is temporarily disabled, to ensure

9 https://vimeo.com/15594590

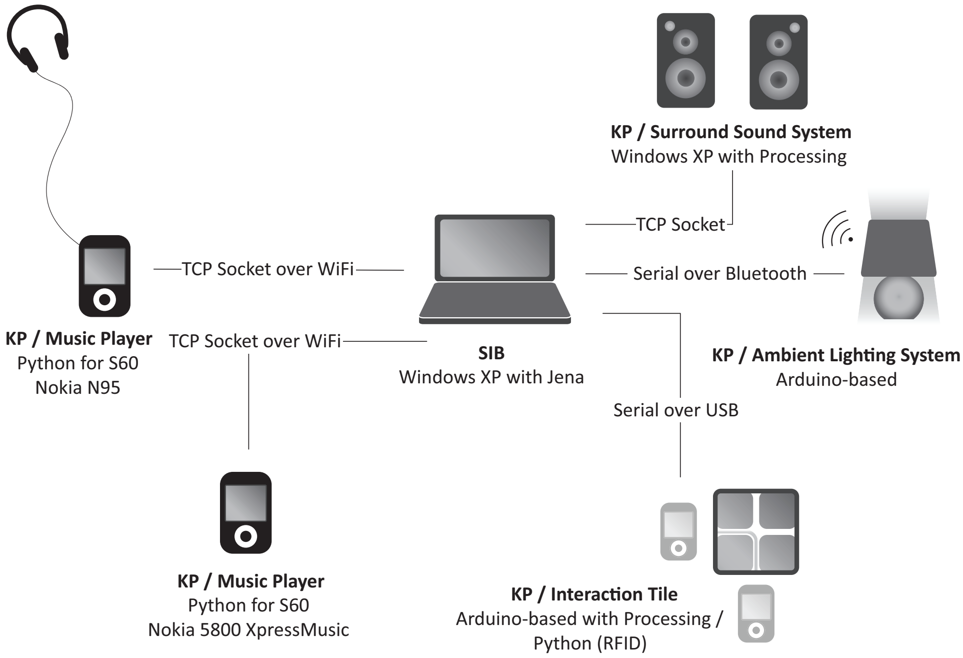


Figure 11: An overview of the demonstrator

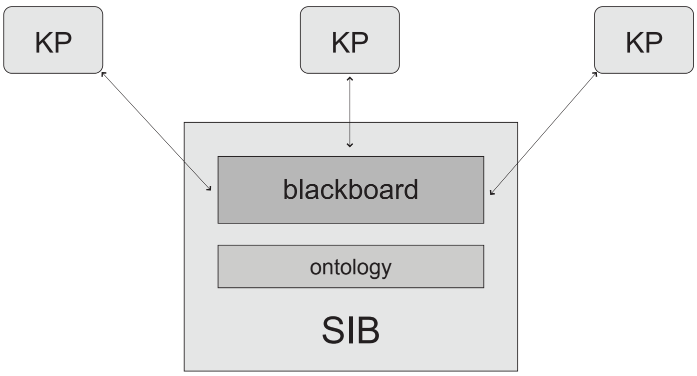


Figure 12: System architecture of demonstrator

that the tag will only be read again after being removed from the field and coming into range again.

If the tile is shaken and a connection is possible, the KP updates the SIB by inserting `connectedTo` relationships between the devices, represented by the cubes next to the tile. If there are existing connections, the `connectedTo` relationships are removed instead. When a cube is removed from the tile, the Arduino again sends the position of the cube via the serial interface to notify the KP. The `python-pyscard`, `pcsc-tools` and `pcscd` PC/SC libraries are required on Ubuntu Linux to communicate with the RFID reader.

3.4.2 *Music Player KP*

This Python-based KP runs on Symbian S60. It has been tested on a Nokia N95 and Nokia 5800 XpressMusic phone. When the KP starts up, it connects to the SIB, generates a `ConnectEvent` and subscribes to new `PlayEvents`, `StopEvents` and `CueEvents`. It then enters an event loop. Pressing the play/stop/forward buttons on the phone's GUI will generate the corresponding event, and the KP will also respond to events generated by other devices that it is connected to via the `connectedTo` relationship.

Another version of the music player KP was developed for a Nokia N900 smartphone that runs on Maemo 5 Linux. This KP was also written in Python and makes use of the PyQt4 library. This KP is functionally equivalent to the Symbian S60 version, apart from running on the Maemo platform and using the Qt4 Phonon framework to provide music play/stop/forward capabilities.

3.4.3 *Light KP*

This KP was written in Java and makes use of the Minim audio library[10] for beat detection, in order to generate meaningful lighting patterns that can be sent to the table lamp.

The Minim audio library is part of the Processing software development environment, used for interaction design prototyping.

The KP listens for media player events from connected devices, and generates RGB values based on the rhythm of the music. These RGB values are then sent to the Arduino in the table lamp via the serial-over-Bluetooth interface. On Ubuntu Linux the `librxtx-java` package is required for serial communication when using Java.

Part of the event handler that handles subscriptions from the SIB is shown in the following code fragment:

```
@Override
public void kpic_SIBEventHandler(String xml) {
String subject = null;
String object = null;
String predicate = null;

println("Subscription notification!");
//Get triples that were added or updated in the SIB
Vector<Vector<String>> triples =
                xmlTools.getNewResultEventTriple(xml);
```

10 http://code.compartmental.net/tools/minim/

```
if(triples!=null){
  for(int i=0; i<triples.size() ; i++ ){
        Vector<String> t=triples.get(i);
        subject=xmlTools.triple_getSubject(t);
        predicate=xmlTools.triple_getPredicate(t);
        object=xmlTools.triple_getObject(t);

        // when we have a new connectedTo
        // relationship to the LightKP
        if(predicate.contains("connectedTo") &&
                        object.contains(deviceID)){

                //subscribe to source events
                subscribeToSourceEvents(subject);
  }
  ...
```

When the Light KP is connected to another device KP using a `connectedTo` relationship, we subscribe to the interaction events generated by that device using the `subscribeToSourceEvents()` function. This code fragment is shown below as an example of how subscriptions are created using the Java KP interface:

```
void subscribeToSourceEvents(String source) {

println("Subscribing to source events from " + source);
xml=kp.subscribeRDF( null , sofia + "launchedBy" , source, URI);

if(xml==null || xml.length()==0){
print("Subscription message NOT valid!\n"); return;}
print("Subscribe confirmed:"+
(this.xmlTools.isSubscriptionConfirmed(xml)?"YES":"NO")+"\n");

if(!this.xmlTools.isSubscriptionConfirmed(xml)){return;}
String sub_id = this.xmlTools.getSubscriptionID(xml);
println("Subscription ID: " +sub_id);
subscriptions.put(source,sub_id);
}
```

3.4.4 *SIB*

The first SIB implementation used in the SOFIA project is called Smart-M3, developed by Nokia, and an open source implementation is available online[11]. The SIB is written in C and uses Nokia's Piglet triple store as a database backend. It is only available on Linux as it makes of the D-Bus message bus system. Other dependencies include the Avahi service discovery framework and Expat XML parser. The SIB consists of a daemon called `sibd`, which communicates with KPs over TCP/IP using a `sib-tcp` connector module.

3.5 DISCUSSION & CONCLUSION

This first iteration constructed a number of devices that could be reused in future iterations, and explored approaches to creating connections between devices. These approaches were focused at proximal interactions with tangible interfaces instead of the usual GUI-based solutions. Let us look at some issues that were uncovered during the implementation, followed by a conclusion.

This iteration details the first use of Smart-M3, where KPs communicate with a SIB using Smart Space Access Protocol (SSAP)[59]. SSAP consists of a number of operations to insert, update and subscribe to information in the SIB. These operations are encoded using XML. A triple-format query from a KP is sent, and the response from the SIB is in triple-format as well.

We attempt to solve the interoperability problem by following a blackboard-based approach. Some of the problems associated with current blackboard-based platforms are scalability and access rights. While the goals of this thesis do not involve solving these problems, they should be considered as possible constraints. In Chapter 11 we will look in more detail at the performance-related issues of the system architecture.

The evaluation of this iteration, where the various alternative tangible approaches are compared in a usability study, is discussed in more detail in [67]. This study was performed in the Context Lab at the Eindhoven University of Technology, and made use of the Teach-Back protocol [115] and Norman's Action Cycle Diagram [84].

From this first iteration we learned that using interaction events to model device and user interaction works well. Yet the different types of interaction events need to be generalised, so that they can be reused in other scenarios and environments. One difficulty we encountered was how to model the capabil-

11 http://sourceforge.net/projects/smart-m3/

ities of devices in more detail so that they can be shared with other devices. In the next chapter we extend the scenario to include devices from our various partners in the SOFIA project, and model the media capabilities of devices in order to perform semantic matching of different media types.

4 DESIGN ITERATION II

If interaction design is considered only at the end, software is driven by engineering design, of which users are rightly unaware, rather than by representations with which they interact.

— Gillian Crampton Smith and Philip Tabor [125], interaction designers

Parts of this chapter appear in [81], [119] and [120].

The second iteration was driven by a collaboration with various partners in the SOFIA project, which included Philips, NXP, Conante and the TU/e System Architecture and Networking (SAN) research group. This collaboration culminated in a joint demonstrator that was exhibited and evaluated at the Experience Lab at the High Tech Campus in Eindhoven — the Smart Home pilot.

The goals of the Smart Home pilot were as follows:

- Conduct a pilot study with users in a setting that resembles a real home
- Demonstrate the system to stakeholders and other interested parties
- Serve as a feasibility study
- Test how stable the implementation would be when it would be running for a full week
- Serve as an experimental setup for user experiments

More general goals of this design iteration were to improve our approach to solve interoperability problems between devices using semantic technologies, and to test our approach to technology integration with an increased number of devices.

4.1 REQUIREMENTS

The Smart Home pilot is based on the following scenario:

Mark and Dries enter their home. A presence sensor detects their presence and notifies the smart space. The decorative wall-wash lights *are in turn notified of user presence by the smart space, and turn themselves on. Mark and Dries start listening to music. They would like to try to render the music on a lighting device to also create some visual effects accompanying the music. They query the smart space and find out that the lighting device can render these light effects. They make a connection between the music player and the lighting device using the Connector. The light starts being rendered on the lighting device. To put the focus on the lighting device, the decorative wall-wash lights in the room automatically dim themselves down. At the same time, the light pattern also starts being rendered on the remote lighting device, where Mark's sister Sofia can observe the same light effects in her own house.*

At another location: Sofia enters her house and the intelligent lighting system detects her presence, notifies the smart space and switches the lights on. After a while, Sofia is curious and wants to listen to the music that Mark and Dries are listening to. She connects her lighting device to her stereo using Spotlight Navigation, and the same song plays on her surround sound system.

There are some obvious similarities with the previous scenario in Chapter 3. However, there are a number of additional devices introduced:

- Presence sensor and wall-wash lighting - A system developed by TU/e SAN that detects the user's presence and switches the lights on automatically
- Lighting device - An ambient lamp developed by Philips, based on their LivingColors technology
- Intelligent lighting system - A lighting system developed by NXP that also detects the presence of users
- Spotlight Navigation - An augmented reality approach to exploring semantic connections, based on technology developed by Conante

This increase in the heterogeneity of devices helps us to see if the ontology-based approach can be applied to a larger range of devices, and the increase in system complexity is used to test the capabilities of the software framework. We first focus on the design of the ontologies that were created to support these kind of scenarios, followed by the design of the devices that were created specifically for the pilot.

4.2 ONTOLOGY DESIGN

The *Semantic Media* ontology and *Semantic Interaction* ontology were created during Iteration II to enable interoperability between the devices of the different partners involved in the Smart Home pilot.

4.2.1 *Semantic Media ontology*

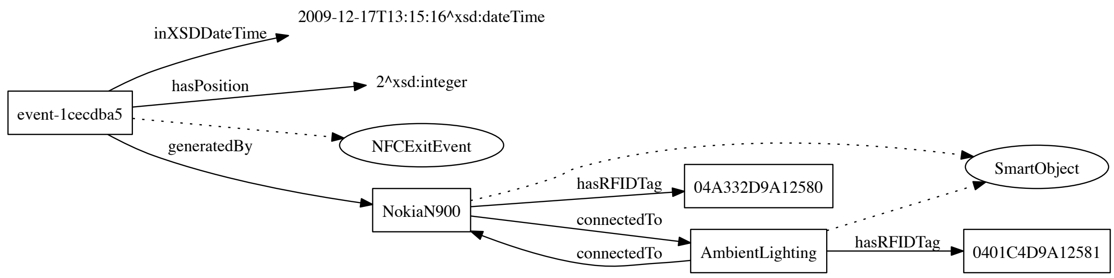


Figure 13: Semantic Media Ontology

The notation used for Figure 13 was also used in Figure 6 in Section 3.2, where class membership is denoted with dotted lines and relationships are denoted with solid lines.

The Semantic Media ontology, shown in Figure 13, is an application ontology that allows for describing media-specific device capabilities and related media content. A mobile device may be described as follows:

```
MobileDevice rdf:type :SmartObject .
MobileDevice acceptsMediaType Audio .
MobileDevice transmitsMediaType Audio .
MobileDevice hasMedia "file://media/groove.mp3"^^xsd:anyURI .
MobileDevice rendersMediaAs Audio .
```

The system configures itself through semantic reasoning based on these media type descriptions. A media player event of type `PlayEvent`, that would be generated when the mobile device starts playing music, is described as follows:

An example of semantic reasoning with media type descriptions is described in Section 4.4.

```
event1234-ABCD rdf:type PlayEvent .
event1234-ABCD inXSDDateTime "2001-10-26T21:32:52"^^xsd:dateTime .
MobileDevice launchesEvent event1234-ABCD .
```

Smart objects may be connected to one another using the `connectedTo` relationship. When a device receives an event notification, it first verifies that it is currently connected to the device that generated the event, before responding to the event.

Smart objects may be connected to one another directly if there is a semantic match between transmitted and accepted media types. Otherwise a *semantic transformer* will have to be introduced to transform the shared content, while still preserving the actual meaning of the connection.

An example of a semantic transformer is described in Section 4.4.

SEMANTIC TRANSFORMER A semantic transformer is defined as a service that transforms information shared between devices from one type to another, while preserving the meaning of the information.

The concept of a semantic transformers is considered an important part of the theory developed in this work, and its applicability to smart environments in general is discussed in Chapter 6.

4.2.2 *Semantic Interaction ontology*

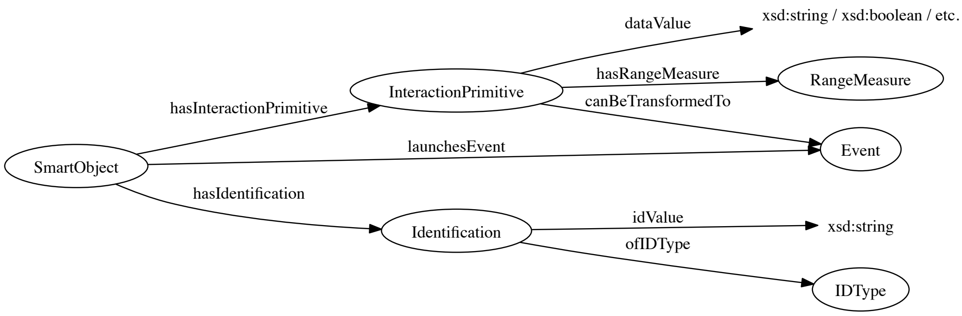


Figure 14: Semantic Interaction Ontology

The Semantic Interaction ontology we have developed is shown in Figure 14. A device, defined as a `SmartObject`, is uniquely identified by some kind of `Identification`, for example a IP address and port number, RFID tag or barcode. Different ID types can be defined as required. Devices can then launch events, for example a media player can generate a `PlayEvent` when music starts playing.

Identification is discussed in Section 6.2.1.

A smart object is described in terms of its *interaction primitives*. This new concept, as well as the other new concepts introduced in this section, refine the ontology that was introduced in the first design iteration:

INTERACTION PRIMITIVE An *interaction primitive* is defined to be the smallest addressable element that has a meaningful relation to the interaction itself.

The concept of interaction primitives is discussed in more detail in Section 6.2.2.

As an example of how the ontology may be used, we start off by defining a smart object and its interaction primitives. Recall that it is only necessary to describe interaction primitives of a device if we use that device's interaction primitive to control another device through the smart space. We can, for example, describe the volume control rocker switch on a smart phone as an interaction primitive:

```
SmartPhone rdf:type SmartObject .
PhoneRockerSwitch rdf:type InteractionPrimitive .
SmartPhone hasInteractionPrimitive PhoneRockerSwitch .
```

We now need to define the properties of the interaction primitive. We start by describing the range measure, or the range of values that the interaction primitive can produce (e.g. the rocker switch can produce `Up`, `Down` or `Neutral` values).

The range of values that an interaction primitive can take on is specified using a `RangeMeasure`. These range measures are similar to the measure of the domain set used by MacKinlay et al [72]. Using the range measures, we can then infer which transformations may be used to map the input values to other interaction primitives or events. The ontology could be extended to also describe the different manipulation operators of the interaction primitive, e.g. rotation on the z-axis or movement along the y-axis. Note that the model of MacKinlay et al. has only been applied to GUIs. Similar models for ubiquitous computing have so far not given comprehensive taxonomies of input devices. Our approach of using interaction primitives to describe input devices is an attempt at providing such a taxonomy.

MacKinlay's work was discussed in Section 2.4.

The actual data value of the interaction primitive is described using the `dataValue` property. Data values may be strings, boolean values or other datatypes, e.g.:

```
PhoneRockerSwitch dataValue "neutral"^^xsd:string .
```

When `PhoneRockerSwitch` is pressed, the data value is updated with:

```
PhoneRockerSwitch dataValue "up"^^xsd:string .
```

This enables other devices to make use of the user input on the `PhoneRockerSwitch`, irrespective of the interaction events generated. In fact, using `Transformation`, it becomes possible to map the physical, generic button presses from interaction primitives like `PhoneRockerSwitch` to specific high-level events like `VolumeUpEvent` or `VolumeDownEvent` using the default transformation `AdjustLevel` as is described in Table 3.

By specifying the transformation using the proper OWL 2 semantics, the reasoner should be able to infer which user inputs can be mapped to which specific high-level events. This shows up as a `canBeTransformedTo` property between an interaction primitive and an event. In our example, this means that the following relationship will be inferred:

```
PhoneRockerSwitch canBeTransformedTo VolumeEvent .
```

where the `"up"` data value may then be mapped to `VolumeUpEvent` and the `"down"` may be mapped to `VolumeDownEvent`, which are both sub-classed from `VolumeEvent`. This prevents situations where arbitrary mappings causes some of the semantics of the interaction to disappear.

4.3 DEVICE DESIGN

In the Smart Home pilot, the partners involved each created their own device or system to showcase the work they have performed during the SOFIA project. The interoperability of the system architecture was tested and demonstrated by having these devices working together, even though they were created by different manufacturers at different times. We now describe these devices in more detail.

4.3.1 *Wall-wash lighting and presence sensors*

The decorative wall-wash lights consisted of four LED lamps, custom-built by the TU/e SAN group, capable of generating coloured illumination on the wall of the room. The lamps are shown in Figure 15, including a description of its components. A presence sensor determines the presence of a user in a designated area of the room and sends the presence information to the SIB. The wall-wash lighting KP is subscribed to this presence information, and its state is modified based on this information. There are two states updated on the SIB: `Away` and `Present`.

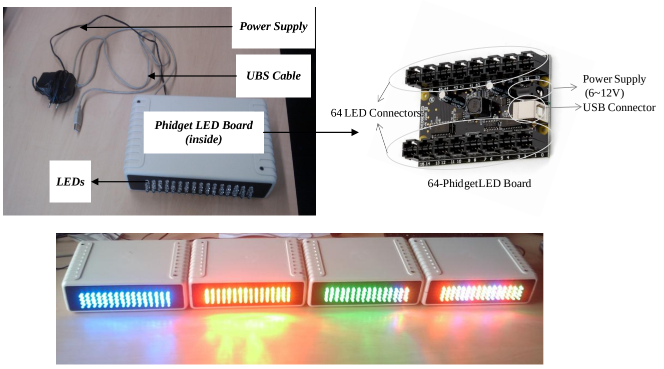


Figure 15: Wall-wash lighting developed by TU/e SAN

Figure 16: The Connector prototype and a smart phone used as a media player

For example, when the `Present` state is specified, the Lamp KP sends the `ON` command to the wall-wash lighting, and the `OFF` command when the `Away` state is specified.

4.3.2 *Connector object*

This device builds on the work done in the previous iteration by exploring another tangible approach for manipulating semantic connections. While devices are still identified with RFID tags, the device itself is now mobile, and makes use of more meaningful interactions and feedback to establish and break connections.

The Connector object, shown in Figure 16, can be used to explore and manipulate semantic connections between different devices in the home environment. It is a handheld device that identifies devices, by scanning RFID tags that are located on the devices themselves. By holding the Connector on top of the tag, users can explore the connection possibilities that are visualised with lights on top of the Connector. After holding the device in the RFID field for a moment, the device-ID is locked and the other device to be connected can be selected in a similar fashion. With a push-to-click action a connection between two devices can be established. For removing an existing connection, the ring on the lower part of the device should be pulled until it clicks.

The cylindrical shape of the Connector is loosely inspired by that of a loupe or hand lens. By moving the connector over a tag, the connection possibilities can be "read" from the top of the cylinder. The display consists of two rings (made up of LEDs), each divided into 4 segments. The Connector supports several actions:

- Explore - You can move it over an object or tag to see whether it is active.
- Select - A device or object can be selected by holding the connector close to or on a tag until the selection sequence is completed.
- Connect/disconnect - The connector can be compressed by pushing the top and the lower part together (connect), and it can be pulled, by pulling the lower part and the top part away from one another until it clicks (disconnect).

When the tag is in the range of the Connector's RFID field, it reads the tag and the first (yellow) light segment on top of the Connector will light up, serving as feedback that the Connector recognises the device. After holding the Connector over a device tag for a moment, a sequence starts, lighting up the second, third and fourth segment of the inner ring. After the device is recognised and selected, another device may be selected in a similar fashion. Now, the second ring of lights will start lighting up in sequence and one should wait until both rings are fully lit. Removing the Connector from the tag prematurely cancels the selection process.

When a connection between the selected devices is possible, both rings start flashing green. When no connection is possible, they will turn red. When a connection between the devices you scanned already exists, the rings will turn green. To make the connection, the Connector is compressed by pushing the top and lower part together, or by pushing the Connector down on the device it is touching, until it clicks. To remove an existing connection between two scanned devices, the ring on the lower part of the Connector should be pulled until it clicks. The rings will show a red light to indicate that the connection has been broken. The segments will turn off once the Connector is moved away from the device. Performing the opposite action of what is required to make or break a connection, cancels the procedure.

The Connector contains the following main components:

- Arduino Stamp 02
- Innovations ID-12 125kHz RFID reader
- SparkFun Bluetooth Mate Gold
- 8 bi-colour LEDs
- Switches
- 3.3v LiPo battery (850 mAh)

In the previous iteration we had an issue with multiple tags within range of the reader at the same time, necessitating the use of 13.56MHz tags. Since only one tag is now read at a time we do not have this issue anymore, and as such 125KHz tags could be used. The Connector prototype is made out of four separate pieces which are 3D printed. The lower part and the top part of the Connector can be moved inward and outward serving as a two-way spring-loaded switch. The prototype packages all the necessary components into one integrated device which is wirelessly connected to a computer using a Bluetooth connection.

4.3.3 *Spotlight Navigation*

The Spotlight Navigation device, designed by Conante and shown in Figure 17, is another approach to explore and manipulate connections between smart devices. With Spotlight Navigation, connection information contained in the smart space is projected into the real world, augmenting the real environment with virtual information, making it intuitively perceivable for users. Spotlight Navigation projects icons close to the actual devices in physical space. It allows for the creation of new connections simply by drawing lines between these icons, using

Figure 17: Spotlight Navigation prototype

a "pick-and-drop" action with a push-button on the prototype (press and hold the button when pointing at one device, move over the second device and release the button). Additionally the connection possibilities are projected between devices that allow for a connection, by changing the colour of the projected line (while the connection is being drawn) from yellow to green when the line's end is moved over the frame of the targeted device. When a connection is impossible, the connecting line will turn red and disappears as soon as the button is released.

Spotlight Navigation was invented as an intuitive way of accessing large data spaces through handheld digital projection devices [95, 120]. Rather than directly projecting the equivalent of a small LCD display, Spotlight Navigation continuously projects a small portion of a much larger virtual pane or data space. It is the device's orientation that defines which part of the larger pane is selected for display. This is done in such a way that the virtual data appears to have a fixed location in the real world. By moving the projector's light spot over the wall, users make portions of the data space visible through intuitive, direct pointing gestures. This intuitiveness stems from the fact that the projected content always stays roughly at the same physical place, regardless of the orientation of the device. It becomes visible depending on whether it is in the projector's light cone or not. In other words, users have the impression that they are illuminating a part of a conceptually unbounded virtual data space, just as if they would be looking at hieroglyphs on a huge wall in a tomb with a flashlight. As people are familiar with operating flashlights, the operation needs no or little training. When accessing a data space with the device, users

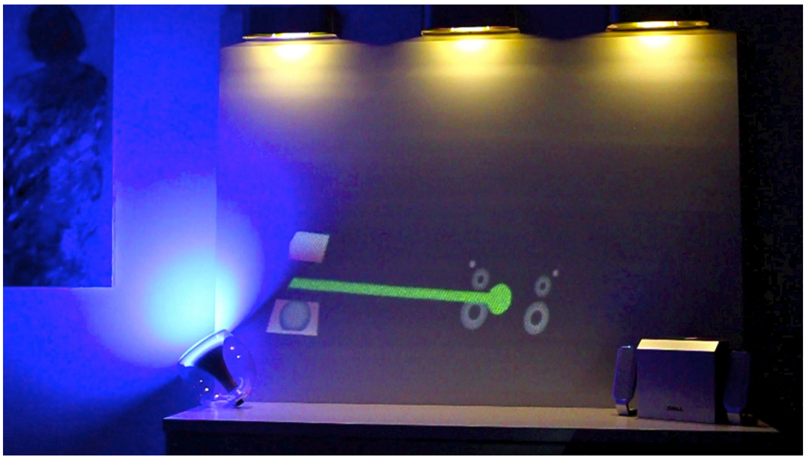


Figure 18: Projection of the Spotlight Navigation when connecting two devices together

can zoom in and out of the data space by using a scroll wheel control, resulting in a pan-and-zoom user interface. To visualise the semantic connections in physical space, we rely on the symbolic meaning of colour, where green colour means "proceed" and red means the opposite. Using green, yellow and red lines we aim at referring to the "existence" of a connection, the "possibility" of a connection or to indicate that a connection is not possible. Figure 18 shows the projection when connecting two devices together.

With Spotlight Navigation, devices are identified by their physical location, relying strongly on *natural mapping*. Connections are created simply by drawing lines between the devices. An erasing gesture with the Spotlight Navigation device pointed at an existing connection, breaks the connection.

On a technical level, the operation is achieved through continuously measuring the orientation, and optionally also the position, of the device. Our prototype is using an inertial navigation module, also called an inertial measurement unit (IMU), that directly measure the orientation by means of accelerometers, gyroscopes and an electronic compass.

The Spotlight Navigation prototype is a fully embedded setup integrated into a 3D printed casing. The design of the casing was targeted at getting the smallest possible setup that could run on the integrated batteries. A dummy ring was added to the prototype to strengthen the semantics of a mobile projector. Figure 17 shows the prototype. Our current setup consists of the following components:

- OMAP3530 board (IGEP module)

Figure 19: Image showing the Connector scanning the lighting device.

- Pico projector (Microvision SHOWWX)
- Orientation sensor (Sparkfun 9DOF Razor IMU)
- scroll wheel (with button press functionality)
- two additional buttons
- two 3.7v li-ion batteries (Nokia BL5J)

The OMAP3530 processor contains a 3D-graphics core (PowerVR) that is capable of rendering the connection visualisations and device icons in real-time. The prototype required the object positions to be manually configured in space, as it did not contain a camera. By using a camera, as is planned for future versions, the intention is to recognise the identity and physical location of each device, so that it is no longer necessary to align the projected object icon with the location of its associated device.

4.3.4 *Lighting Device*

Philips created two lighting devices based on their LivingColors technology, that can be used to generate dynamic coloured lighting. Figure 19 shows the Connector object scanning a tag on the lighting device. These lighting devices accept a stream of RGB values and use the information to generate a sequence of coloured lighting. Using the media type descriptions introduced above, we can describe a lighting device as follows:

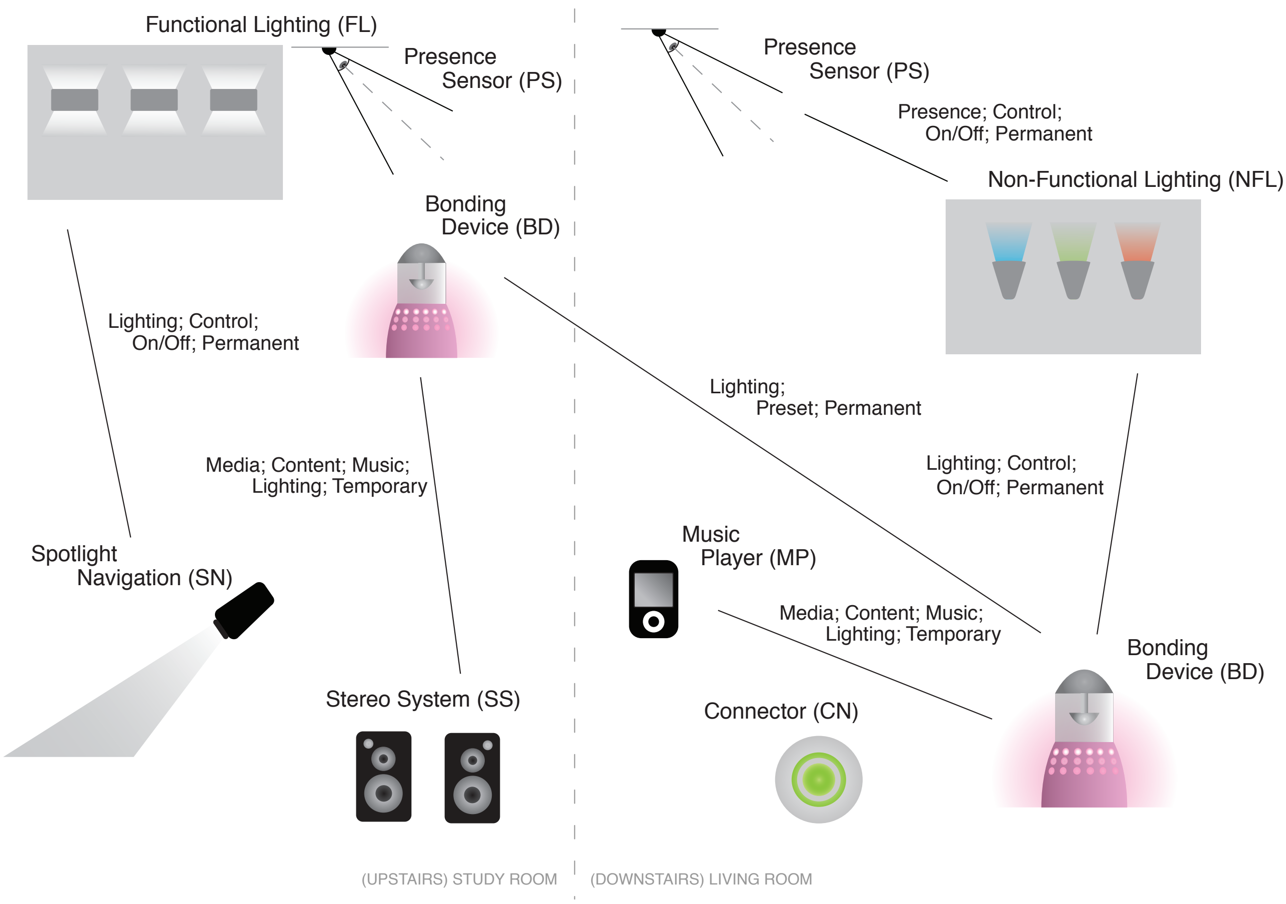


Figure 20: The devices and their connections as used in the system

```
LightingDevice rdf:type SmartObject .
LightingDevice acceptsMediaType RGBValues .
LightingDevice rendersMediaAs Lighting .
```

In the scenario there exists a permanent semantic connection between the two lighting devices. This means that when dynamic lighting is generated on one device, the same lighting will be displayed on the other device.

4.4 IMPLEMENTATION

Figure 20 shows a brief overview of the different parts of the system. The experiment took place in two rooms, the study and the living room of the Experience Lab on the High Tech Campus in Eindhoven. During the pilot, users interacted with various automated and interactive appliances and devices defined as smart objects. There exist several semantic connections between the smart objects, for example the media-content connection between the phone and the lighting device, and the lighting-control connection between the lighting device and the non-functional lighting. Some of these connections can be explicitly interacted with through two interaction devices: a Spotlight Navigation device placed in the study of the pilot setup upstairs, or a Connector device placed in the living room of the pilot setup downstairs.

In the Smart Home pilot, media content is shared among several smart objects in a smart home setting. Music can be shared between a mobile device, a stereo speaker set and a lighting device that can render the mood of the music with coloured lighting. The music experience is also shared remotely between friends living in separate homes through the lighting device. This light and music information is shared between the two lighting devices. Other lighting sources, like the smart functional lighting (FL, Figure 20) and the smart wall wash lights (NFL, Figure 20) are sensitive to user presence and the use of other lighting sources in the environment. The setup was built using the SOFIA software platform as is described in Chapter 10. A diagram showing the technical details of the Smart Home pilot is shown in Figure 21. It gives an indication of the variety and complexity of the hardware platforms, operating systems and wireless protocols that were used.

4.4.1 *ADK-SIB*

The Jena framework was first mentioned in Section 3.2.

In this iteration a new SIB developed within the SOFIA project was used, called the ADK-SIB. The ADK-SIB is a Jena-based SIB written in Java and runs on the Open Services Gateway initiative (OSGi) framework. Some modifications were made to the standard ADK-SIB provided by the SOFIA project, such as reasoning support added with the TopBraid SPIN API 1.2.0[1]. To run the SIB from the OSGi prompt, the SIB and TCP/IP gateway is started separately as services:

```
sspace create -sib -name=test
sspace create -gw -name=testgw -type=TCP/IP -idSib=1
sspace start -sib -id=1
sspace start -gw -id=1
```

The SIB and gateway are linked with one another through their IDs, enabling multiple SIBs and gateways to run on the same machine. OSGi services have to be deployed as plugins from within the Eclipse development framework.

1 http://topbraid.org/spin/api/

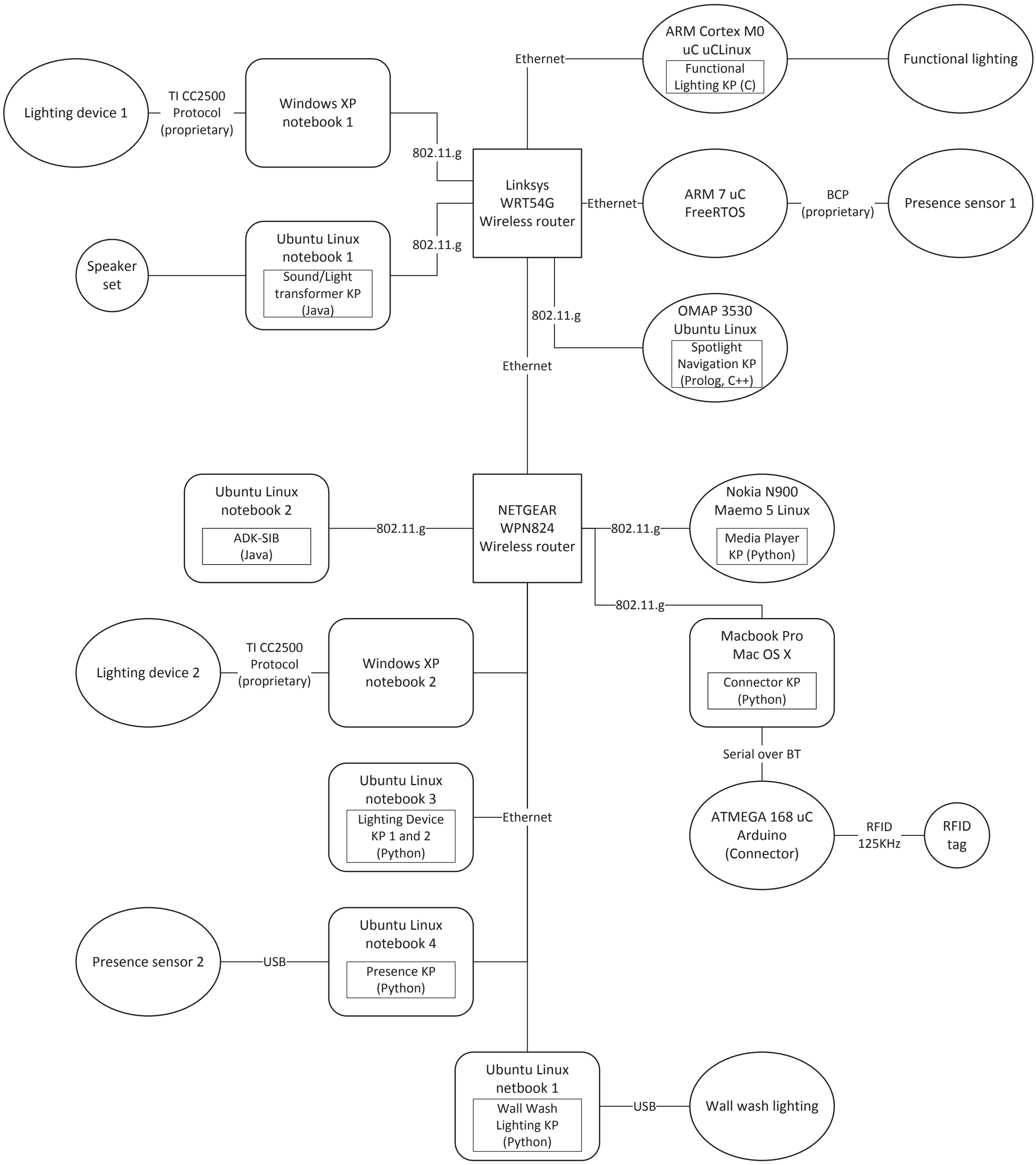


Figure 21: Technical details of the Smart Home pilot

More details on the different types of reasoning that was performed, as well as the different services provided by the reasoner, are available in Section 9.3.

Reasoning on information contained within the SIB was performed using SPARQL Inferencing Notation (SPIN)[2]. With SPIN, rules are expressed in SPARQL, the W3C recommended Resource Description Framework (RDF) query language, which allows for the creation of new individuals using CONSTRUCT queries. OWL inferences for the OWL 2 Rule Language (RL) profile were executed by using SPIN rules[3]. OWL 2 RL is a syntactic subset of OWL 2 that is amenable to implementation using rule-based technologies. According to the OWL 2 RL W3C page[4] the OWL 2 RL profile is aimed at applications that require scalable reasoning without sacrificing too much expressive power.

4.4.2 *Semantic matching of media types*

A semantic transformer, called the Sound/Light KP, accepts a music stream as input and generates a stream of RGB values based on an analysis of the music stream. The Sound/Light KP is described as follows:

```
SoundLightKP rdf:type SemanticTransformer .
SoundLightKP acceptsMediaType Audio .
SoundLightKP transmitsMediaType RGBValues .
SoundLightKP  hasIdentification id4321 .
id4321 ofIDType IPAddress .
id4321 idValue "192.168.1.4:1234" .
```

The stream of RGB values is sent via a separate TCP/IP connection, so the lighting device needs to know whether the source device is capable of communicating via TCP/IP. Since smart objects in the smart space can be identified using their IP address and port number, we can use the identification information to infer a `communicatesByTCPIP` data property that can be read by the Bonding Device. To relate the `SmartObject` directly to the `IDType`, we use an OWL 2 property chain:

$$\text{hasIdentification} \circ \text{ofIDType} \sqsubseteq \text{hasIDType}$$ [5]

We then infer the `communicatesByTCPIP` data property by specifying a `TCPIPObject` subclass:

---
2 http://www.spinrdf.org
3 http://topbraid.org/spin/owlrl-all
4 http://www.w3.org/TR/owl2-profiles/#OWL_2_RL
5 The concatenation of two relations R and S is expressible by $R \circ S$, while $R \sqsubseteq S$ indicates that R is a subset of S

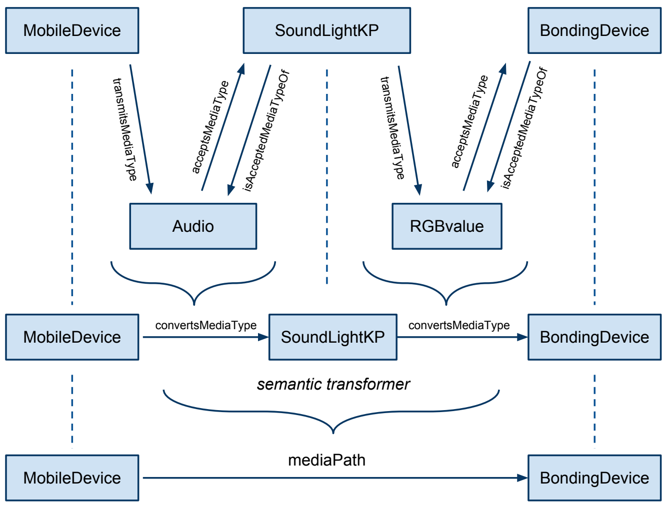


Figure 22: Inferring the media path

```
Class: TCPIPObject

    EquivalentTo:
        hasIDType value IPAddress
        communicatesbyTCPIP value true

    SubClassOf:
        SmartObject
```

In order to determine the media source for the lighting device, we first need to perform semantic matching of the media type descriptions. We first define `isAcceptedMediaTypeOf` as the inverse property of `acceptsMediaType`, and then define the following property chain:

transmitsMediaType ∘ isAcceptedMediaTypeOf ⊑ convertsMediaType

This allows us to match media types between smart objects. We can then infer a *media path* between the mobile device and the Bonding Device with the Sound/Light KP acting as a semantic transformer using another property chain:

convertsMediaType ∘ convertsMediaType ⊑ mediaPath

To then determine the media source itself we use Semantic Web Rule Language (SWRL)[6], as the expressivity of OWL does not allow for inferring the media source if there are more than one

6 http://www.w3.org/Submission/SWRL/

`convertsMediaType` relationship linked to the lighting device:

```
convertsMediaType(?x1,?x2) ∧ convertsMediaType(?x2,?x3) ⇒ mediaSource(?x3, ?x2)
```

The media source is the semantic transformer, `?x2`, while the media path is between the two smart objects, `?x1` and `?x3`. The `mediaSource` relationship is thus inferred from the smart object to the semantic transformer. We can also infer whether a device is a semantic transformer or not using:

This SWRL implementation was later replaced using another Semantic Web technology called SPIN, detailed in Chapter 9.

```
Class: SemanticTransformer

    EquivalentTo:
        (canAcceptMediaTypeFrom some SmartObject) and
        (convertsMediaType some SmartObject)

    SubClassOf:
        SmartObject
```

The end result is that the lighting device responds to the mobile device's media events (based on the Semantic Connections `connectedTo` relationship), but uses the Sound/Light KP as a media source for generating dynamic lighting. The `connectedTo` relationship between the mobile device and the lighting device should only be possible if a media path exists between the two devices. Figure 22 illustrates the entire process of inferring the media path from the original media type definitions.

If the reasoner infers a media path between two smart objects, it does not mean that they are automatically connected – it means that a connection is possible. The user can view this connection possibility using either the Connector device or the Spotlight Navigation device, and then establish the connection if necessary.

4.4.3 *Device states*

Interaction events (Chapter 8) cause device state changes. Most of the developers that worked on the Smart Home Pilot preferred to describe their smart objects in terms of the device states, and also shared these device states with other smart objects using the SIB. The current state of the smart object was defined using the `sofia:isInState` property:

```
conante:spotlight1 sofia:isInState "projecting" .
sofia:nflKP1234 sofia:isInState "lightingON" .
```

```
sofia:nflKP5678 sofia:isInState "lightingOFF" .
sofia:presenceKP1234 sofia:isInState "Away" .
sofia:presenceKP5678 sofia:isInState "Present" .
```

These smart objects were all simple two-state devices, where the device state was indicated using a text field. Note that `conante:spotlight1` used the absence of `sofia:isInState` property to indicate that it was not projecting. This statement is valid with a Closed World Assumption (CWA), the presumption that what is not currently known to be true, is false. The programmer that created this state description is well versed in the Prolog programming language, which makes the CWA. OWL, on the other hand, operates under the Open World Assumption (OWA). With OWA, we assume that new information can become available at any time, so that we cannot draw conclusions based on the assumption that all information is already available [3]. We can use the ontology to restrict how state descriptions are reported, forcing smart objects to report their current state at all times.

The OWA is also discussed in Section 9.3.5.

4.5 EVALUATION

A number of issues were identified during a user study of the Smart Home pilot, described in more detail in [120]. In the Smart Home pilot there were two locations connected via a permanent semantic connection between the lighting devices. What if Sofia were to play a song in her room — will the same song play back at the home of Mark and Dries? If this is the case, we clearly need to introduce a notion of directionality in the semantic connections. This issue is addressed in Design Iteration III in the next chapter.

Early performance tests indicated that the Pellet reasoning engine with SWRL rules proved to be a performance bottleneck in the system. For example, Pellet took about 3 seconds to infer 107 statements. TopBraid Composer's TopSPIN reasoning engine supports SPIN rules and OWL 2, so it was tested as a possible alternative. The TopSPIN engine with OWL 2 RL/RDF Rules took less than a second to infer 10 491 triples. By using a hashmap to store our inferred triples, we were able to improve performance even further. Some of the inferred triples were redundant inferences – by using a hashmap we were able to reduce the number of inferred triples on startup from 10 491 to 5 122, eliminating redundant triples.

In one instance, a SWRL rule took up to 28 seconds to execute.

A more formal performance evaluation as well as a user evaluation of the ontology, both of which were performed on the work done in this iteration, is discussed in Chapter 11.

4.6 DISCUSSION & CONCLUSION

Modelling constraints in the ontology is done using *restrictions*. When modelling concepts in an OWL ontology, restrictions are defined either as part of `rdfs:subClassOf` or as part of `owl:equivalentClass`. There is a subtle difference, and it has to do with *necessary and sufficient conditions*.

When we have necessary and sufficient conditions (also known as *if and only if* and denoted as $\equiv$, $\leftrightarrow$ or $\Leftrightarrow$), the `owl:equivalentClass` restriction (denoted as $\equiv$) is used. When we only have necessary conditions, the `rdfs:subClassOf` restriction (denoted as $\sqsubseteq$) is used. *Necessary and sufficient* means that the restriction is sufficiently constrained that only individuals belonging to that class will be classified as such.[7] An example is shown in Section 4.4.2.

An OWL reasoner follows a bottom-up approach, where new information is inferred from asserted facts, compared to a theorem prover that starts from its goal.

The ontology supports the description of interaction data generated by interaction devices and sensors. Additionally, it shows that an interaction primitive may trigger an interaction event or a state change that may need to be specified in more detail by a more application-specific ontology. That is to say, this ontology may also be used to perform semantic mapping from the interaction data to user goals and/or available services [80]. Any additional information related to the smart object may be added by extending the schema defined in the Semantic Interaction Ontology.

Another advantage of the ontology described in this section is that it opens up the way to context-based interaction device reconfiguration. For example, if a Context Monitor application recognises a situation where the `PhoneRockerSwitch` should no longer control the volume, but adjust the level of lighting instead, the triple could be modified accordingly. Just such a simple change would implement a behaviour that adapts to the situation.

Context-dependent functionality changes of a control may not necessarily be a desirable feature. It should however be noted that we only consider context-dependent meaning change with generic interaction primitives, that in itself do not have

7 In the Protégé ontology editor these are also called *defined classes*.

a specific, function related meaning (and might already being used for different functions, like the rocker switch in the example). Additionally, the re-mapping is only considered for those interaction elements with compatible transformational properties, e.g. the rocker switch may only be mapped to other `AdjustLevel` transformations, and not to `Start/Stop`. The specified range measures are used to control the re-mapping between an interaction primitive and an interaction event, in a similar way that the input and output domains of [72] are used to control the expressiveness between an input device and its application parameter.

The question then becomes how to inform the user of the remapping in a user-friendly way. In the next chapter we consider the different types of feedback that can be used. Besides automatic context-dependent functionality changes of controls, we especially consider user-initiated re-mapping of controls. By enabling users to make associations, or semantic connections [117] between devices or interaction elements and devices, users can express their intentions in terms of mapping controls between devices [80].

Judging from the experience of implementing the semantic transformers, the approach of using them to solve interoperability problems appears promising. Using the Semantic Media Ontology, we were able to define a smart object in terms of the media types it accepts and transmits. Based on these descriptions, semantic transformers can be used to transform media types in order to enable information exchange between devices that would normally not be able to communicate. With only a minimal set of device capabilities described, the system is able to perform self-configuration using semantic reasoning.

Even though the Semantic Interaction Ontology describes parts of a `SmartObject`, it does not fully describe all the properties and capabilities of the smart object. It only describes its interaction-related properties. Particularly it defines the `SmartObject` interaction primitives and means of identification. In the next chapter, while describing the next iteration, we will focus on expanding the possibilities of describing a device's functionality and capabilities.

5 DESIGN ITERATION III

Making everything visible is great when you only have twenty things. When you have twenty thousand, it only adds to the confusion.

— Don Norman [85], cognitive scientist and designer

The goal of the final iteration was to extend the scenarios developed in the previous iterations to a new domain, while still making use of the smart objects and concepts that have been developed thus far. This would allow for testing the general applicability of the concepts and techniques, while still being able to reuse some of the devices we have already developed.

5.1 REQUIREMENTS

The use case scenario in this iteration revolves around a person's evening routine before falling asleep. It is a cross-domain scenario that extends the media domain into the sleep domain, and enables the exchange of different types of information. The domain of sleep was chosen for several reasons:

- Sleep is important for physical and mental well-being — an important application area of our research group at TU/e.
- The sleep domain is targeted by a number of recent Internet of Things (IoT) devices that record and share data and can be accessed through their APIs.
- The sleep domain allows us to reuse some of our existing work on media sharing and lighting, extending it into a new domain.

In the fitness and sleep domains there are a plethora of devices that are well-known to the IoT community but that are not interoperable, such as:

- the Withings WiFi body scale[1], that transfers body weight wirelessly to a computer or mobile device,

1 http://www.withings.com/en/bodyscale

- the Fitbit[2] and Nike FuelBand[3] fitness monitors, that track activities using a built-in accelerometer, and
- the Zeo sleep monitor[4], that records sleep cycles using a head-mounted sensor.

Existing software applications targeted at these devices visualise the data coming from these devices. Our goal is to enable serendipitous interoperability, and we are interested in seeing what will happen when the data and capabilities are shared between these devices. For example, we could use the data coming from a sleep monitor to change the behaviour of a light in the room, or the alarm on a mobile phone. We distinguish between a number of subdomains within the area of well-being, as shown in Figure 23.

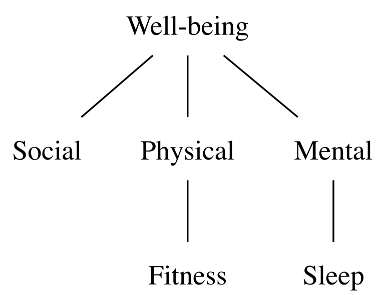


Figure 23: Sub-domains of well-being

Several devices were used in Iteration III, including:

- an Android smart phone – Samsung Nexus S;
- an internet radio – Logitech Squeezebox Radio;
- the lamp from Section 3.3.2;
- a sleep monitor – Zeo Sleep Manager; and
- an Android tablet – Samsung Galaxy Tab 10.1 WiFi.

2 http://www.fitbit.com/
3 http://www.nike.com/fuelband/
4 http://www.myzeo.com

We purposefully did not define a narrative for this design iteration, to refrain from only implementing the functionality described in the narrative. Instead, we looked at the meaningful ensembles we could create with the devices, attempting to allow for *emergent functionalities* to surface by sharing device capabilities and interaction events.

The design iteration was implemented in the master bedroom of the Context Lab of TU/e, a lab with a setting that resembles a real home. Implementing the setup in an environment that allowed us to see its behaviour and implications in a realistic setting, gave insights that are regarded more valuable than obtained when building a setup on for example one's office desk.

5.2 ONTOLOGY DESIGN

In this iteration the earlier ontologies were consolidated into a single ontology. This helps make the ontology more manageable and removes the "cruft" of legacy statements that build up over time.

The first design decision of this iteration was to introduce the notion of directionality. This gives additional meaning to the devices, which now need to be modelled as *sources*, *sinks* or *bridges*. A music player is an example of a smart object that acts as a source when connected to a speaker, which in turn acts as a sink. A smart object can act as both a source and sink, which we define as a bridge. For example, consider the case where the speaker is connected to another speaker, which then also plays back the same music. The first speaker then acts as a bridge.

We can infer that a smart object is a sink using

$$\text{Sink} \equiv \text{SmartObject} \sqcap (\text{functionalitySink} \ \exists \ \text{Functionality})$$

where the symbol $\exists$ is used to denote the existential restriction that `functionalitySink` is some kind of `Functionality`. A bridge is inferred using

See Table 5 on page 142 for more details on the symbols and syntaxes used in this thesis.

$$\text{Bridge} \equiv \text{Sink} \sqcap \text{Source}$$

A semantic transformer is a virtual component that is not physically addressable and is therefore not considered to be a smart object. However, it is a bridge, as it acts as both a source

Figure 24: Logitech Squeezebox Radio

and a sink. A smart object is a physical object first, with a digital representation added later.

Other areas where the ontology was improved include the modelling of device capabilities (Chapter 7) and the modelling of events (Chapter 8). These improvements are discussed in more detail in the relevant chapters.

Semantic transformers were first introduced in Section 4.2 and are discussed in more detail in Section 6.4.

5.3 DEVICE MODIFICATIONS

In this iteration, we reused both the ambient lighting system from Section 3.3.2, as well as the Connector object from Section 4.3.2. For the Squeezebox radio and Android devices new KP software was developed.

5.3.1 *Squeezebox radio*

The Squeezebox radio, shown in Figure 24, can be controlled via a Telnet interface over WiFi.[5]. For example, the accepted parameters for setting an alarm are shown in Table 2.

On startup, the Squeezebox KP connects to the smart space, registers the capabilities of the device, checks for existing connections and listens for new connections. It also subscribes to new system events. It then connects to the Squeezebox device

System events are discussed in more detail in Chapter 9.

5 On Squeezebox Server, the interface documentation is available from `Help` ⇒ `Technical Information` ⇒ `The Squeezebox Server Command Line Interface`

PARAMETER	DESCRIPTION
`dow`	Day of week (0 – 6, starts on Sunday)
`time`	Time since midnight in seconds
`repeat`	1 or 0
`volume`	0 – 100
`url`	Squeezebox Server URL of alarm playlist
`id`	The ID of an existing alarm (optional for new alarms)

Table 2: Accepted parameters for Squeezebox `alarm` Telnet command

via the Telnet-over-WiFi interface, subscribes to new events generated by the device and enters an event loop.

When a new alarm is set on the device, the KP converts the date and time to XML Schema Definition (XSD) format and generates a new `AlarmSetEvent`. When an alarm is triggered, an `AlarmAlertEvent` is generated. If the alarm is dismissed on the device, an `AlarmEndEvent` is generated. When an alarm is deleted, an `AlarmRemoveEvent` is generated.

When an `AlarmSetEvent`, `AlarmRemoveEvent`, `AlarmEndEvent` is received from another device, the corresponding action is performed on the device. The device also responds to media events like `PlayEvent`, `PauseEvent` and `PlayEvent`.

Media events were introduced in Section 3.4.2.

5.3.2 *Android mobile devices*

The KP developed for the Android devices was tested on both the Google Nexus S phone and the Samsung Galaxy Tab.

To improve software reuse and not reinvent the wheel, we wanted to make use of the stock applications on the phone, like the Clock app and the Music app (shown in Figure 25), instead of developing our own. On Android, it is possible to run a service as a background process that listens for events generated by other applications. A *broadcast receiver* listens for *broadcast intents*, which are public intents broadcast from activities to registered receivers. A receiver registers for a broadcast intent by listing it in its intent filter in the manifest file. Broadcast intents sent by Android applications can be received by all other applications, which is done by creating a broadcast receiver.

Android activities run inside applications.

When the alarm is triggered in the alarm app on the mobile phone, a

```
com.android.deskclock.ALARM_ALERT
```

broadcast intent is generated.

Figure 25: Playing music from the phone on the Squeezebox radio

A broadcast receiver handles such an intent using

```
@Override
protected void handleBroadcastIntent(Intent broadcastIntent) {
  String action = broadcastIntent.getAction();
  if(action.equals("android.intent.action.ALARM_ALERT")) {
        addEvent("AlarmAlertEvent");
  }
}
```

The alarm app on the Google Nexus S phone is called `DeskClock` and was developed by Google. There also exists a version for earlier Google phones called `AlarmClock`.

Note that this intent is not supported by all Android devices, as different devices may have different default alarm applications. It did, however, work on both the Google Nexus S phones and Samsung Galaxy tablets that we tested. To determine when an alarm was changed, we made use of the

```
android.intent.action.ALARM_CHANGED
```

broadcast intents. It is also possible to read the next alarm that will triggered from the system settings, using

```
System.Settings.NEXT_ALARM_FORMATTED
```

To determine if a song is being played using the Android Music app, we used the

```
com.android.music.playstatechanged
```

broadcast intent.

We used a Lighted Greenroom [65] pattern to launch a long-running service from a broadcast receiver, without the operating system throwing an Application Not Responding (ANR) message. ANR specifies a 10-second response limit for a broadcast receiver, after which it is deemed unresponsive. By launching a separate service that handles generation of events based on broadcast intents, we have a workaround to this problem. This allows us to listen for broadcast intents from applications like the music player and the alarm clock.

The Lighted Greenroom pattern was introduced by Komatineni et al. [65] to simplify interacting with the Android wake lock.

5.3.3 *Wakeup experience service*

In the sleep use case, music can be shared between the smart phone and the internet radio. Alarms can be shared between the phone and the internet radio, the internet radio and the lamp as well as the phone and the lamp. Because the lamp has only `LightOn/LightOff` and `AdjustLevel` capabilities, the most basic functionality of the lamp responding to an `AlarmEvent`, would be to turn on at the time that the event occurs. However, a wakeup service can be connected that *transforms* an `AlarmSetEvent` into a wakeup experience, sending a sequence of `AdjustLevelEvents` to the lamp. This wakeup service then functions as a semantic transformer, transforming one type of value into another in a meaningful way. Semantic transformers are virtual entities and therefore they do not have a physical presence, in contrast to smart objects that must have a physical representation. Therefore, the use of a semantic transformer is automatically inferred based on its capabilities, as it cannot be physically connected to other devices by the user.

To create a wakeup service, an `AlarmSetEvent` would have to trigger an `AdjustLevelEvent` event with a `dataValue` that increases from 0 to 100 over a period of 30 minutes *before* the alarm sounds. Another requirement is that it should work with any light and any alarm in the smart environment.

This wakeup service has similar functionalities as a Wakeup Light (e.g. as sold by Philips[6]) which means it starts increasing its light level over a 30 minute time-period, reaching full intensity (as calibrated) at the set alarm time. The semantic connection between the phone's alarm and the dimmable light is an example of how such a connection can have emerging func-

6 http://www.philips.co.uk/c/wake-up-light/38751/cat/

Figure 26: The sleep use case scenario, with the Zeo sleep monitor on the left, the dimmable light and the Connector object in the middle, and the Squeezebox on the right

tionality, which does not exist without the connection and the wakeup service.

This opens up many possibilities for users, as they may connect other lights, and potentially even other devices such as a networked thermostat, to either the alarm or the dimmable lamp, creating their own wakeup experience. Whether such emerging functionalities are possible obviously depends on the way the smart objects are implemented. For example in our implementation, the dimmable lamp is described as a sink, which means it is only capable of accepting input. If it was described as a source as well, sharing for instance its on/off state or its current light value, it could act as a bridge and allow for more interesting configurations.

5.3.4 *Zeo*

The Zeo sleep monitor is shown on the left-hand side of Figure 26. The Zeo headband, shown in Figure 27, uses three silver conductive fabric sensors to collect Electroencephalography (EEG) signals while a person is sleeping. The signals are amplified and features are extracted using a Fast Fourier Transform (FFT). An Artificial Neural Network (ANN) is then used to estimate the probability of a person being in a certain phase of sleep[99]. The sleep stages are Awake, Rapid Eye Movement (REM) Sleep, Light Sleep, Deep Sleep or Undefined.

Figure 27: The Zeo headband

Sleep data is stored on an SD card on the device and can be uploaded to the Zeo MySleep[7] website. Zeo created the Data Decoder Card library[8] that allows developers to decode the sleep data without uploading the data to the MySleep website. We also built a USB cable that connects to a serial port on the back of the device. With this cable you can access the raw data coming from the headband sensor.

When the device is connected via a USB cable, we have real-time access to the generated events. Events that could be interesting to other smart objects in the environment include:

- `NightStart` - time when first "Awake" hypnogram occurs
- `SleepOnset`
- `HeadbandDocked` and `HeadbandUndocked`
- `AlarmOff`, `AlarmSnooze` and `AlarmPlay`
- `NightEnd`

A KP was developed for the Zeo sleep monitor. Although it was not used as part of the final scenario, the Zeo shares its data like sleep states and alarm events in the smart space, which can be used by other devices.

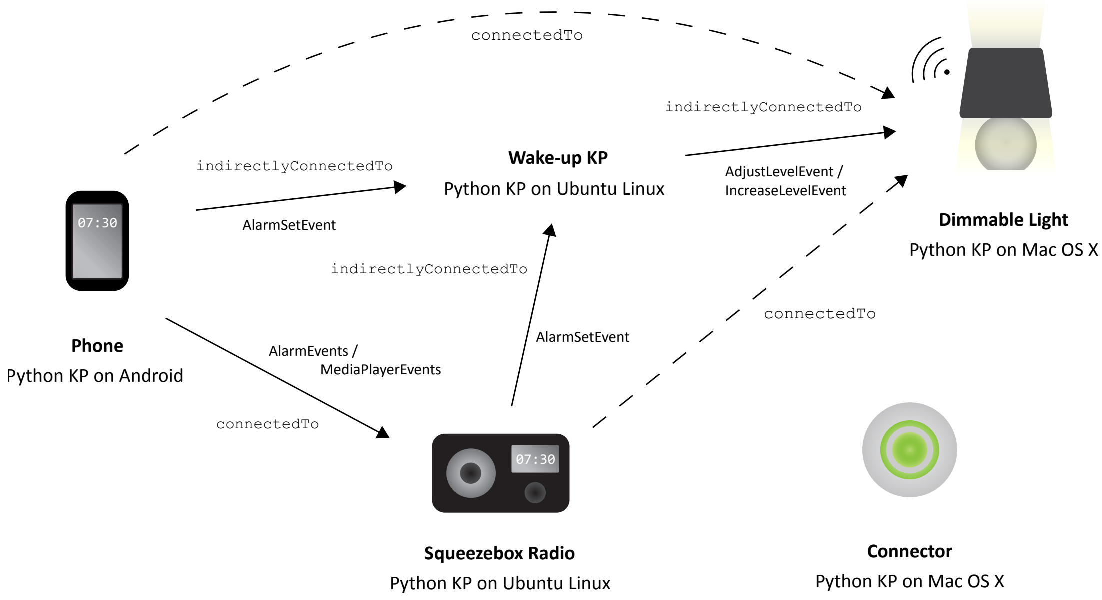


Figure 28: An overview of the sleep use case

5.4 IMPLEMENTATION

In the sleep use case we did not make use of the Zeo and its KP implementation. It was viewed as a backup device which could be used to introduce additional complexity to the system if necessary.

We started by implementing a very basic configuration, connecting the phone and the internet radio. Based on the capabilities of the devices, possible connections included sharing music player functionality and alarm clock functionality. After implementing the first basic functionalities, we gradually increased complexity by adding another smart object, the dimmable lamp, followed by the implementation of several types of interaction feedback.

5.4.1 *Feedback and Feedforward*

In Van der Vlist's thesis [116] there is a similar discussion on feedback, that focuses on some of the more user-centred aspects.

In Section 2.5.1, where the Frogger framework of Wensveen [123] was introduced, we also introduced the terminology of augmented and functional feedback/feedforward. We use feedforward to display a device's functional possibilities. We can use feedback to confirm user actions, using augmented feedback where direct functional feedback is not available.

When an alarm is set on the phone, augmented feedback should be given on all devices connected to the phone. For

7 http://mysleep.myzeo.com
8 http://developers.myzeo.com/data-decoder-library/

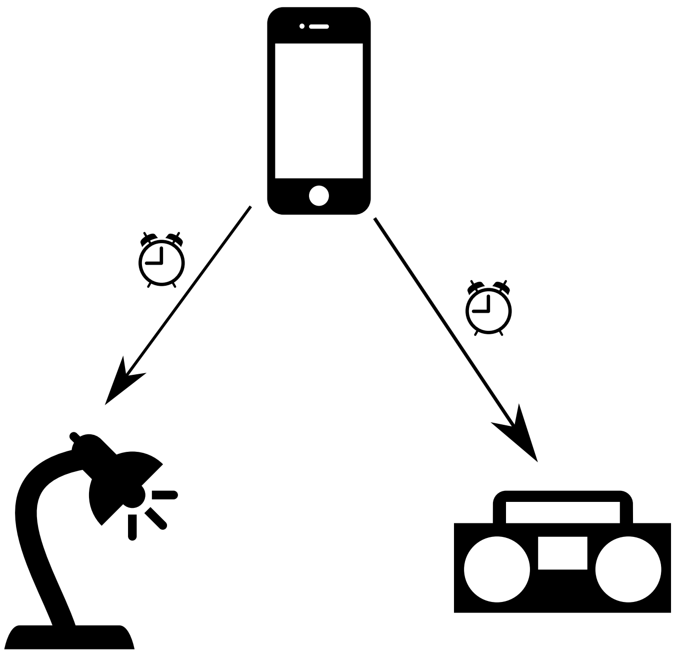


Figure 29: Alarm functionality of the phone shared with the radio and the lamp

example, consider the setup in Figure 29, where the alarm is connected to both the lamp and the Squeezebox radio.

Immediate feedback only makes sense when the event and its feedback coincide in time and modality (e.g. audio, visual). When the generated event is a `SetEvent`, the event itself will occur sometime in the future, so we generate the functional feedforward as augmented feedback instead. For example, for an `AlarmSetEvent` we generate a `1s` alert sound on the Squeezebox radio as augmented feedback, providing functional feedforward of what will happen when the alarm is triggered. We also provide visual augmented feedback by displaying a popup message on the display for a few seconds. On the lamp feedback is given in the form of a short light pulse to confirm that it has been notified as well.

Feedback and feedforward need to be carefully designed when smart objects are interconnected. However, as the smart objects themselves are unaware of each other and, at development time, their designers cannot anticipate what other devices users may connect the smart objects to, the total user experience cannot easily be designed. In this section we will describe how feedback and feedforward were used to enhance the user experience and enable devices that are in-fact unaware of one another, appear to show awareness of each other to their users.

Many solutions for interconnecting devices often employ the vendor lock-in *strategy, which enables manufacturers to have full control over their ecosystem of products and the resulting user experience.*

5.4.1.1 *Augmented and functional feedforward*

For semantic connections, functional feedback and feedforward can only be considered for the combination of source and sink. The source object has functional feedforward that may communicate its function. Only when both the source and sink object have been identified, is functional feedforward available for the semantic connection. Important to note is, that functional feedforward is derived from the intersection of functionalities of both the source and the sink. These functionalities could be ambiguous, as both source and sink may be multifunctional. If this is the case, users should make explicit what information or data they want to exchange by selecting the desired mode on the source object (e.g. selecting the alarm application on your smart phone to share the alarm time or go to a picture viewer when pictures should be exchanged), restricting the possibilities. If this is not possible, or a multifunctional smart object is connected when it is in idle mode, semantic reasoning could be used to match all meaningful capabilities of the source and sink objects.

Whenever users wish to make a connection, they have certain expectations. We can employ functional feedforward to influence these expectations. Additionally, we can enhance the user's understanding by explicitly adding augmented feedforward (i.e. augmented *functional* feedforward in contrast to augmented *inherent* feedforward). In the sleep use-case we employed augmented feedforward in the process of exploring connection possibilities i.e. before the connection is made. We do this by giving a *functional preview* on the sink object, viewing the functionality of the connection that is currently explored. Our reasoning is, that only when both source and sink are identified, we can speak of a semantic *connection* and, by giving the feedforward at the sink, we ensure that the sink object is in fact capable of producing this feedforward (i.e. has the necessary capabilities). Additionally the location of the feedforward corresponds to the location where the action (identifying the sink object) was performed. To do so, a `PreviewEvent` is generated when a possible connection is being explored, displaying the possible functionalities enabled by the connection.

Example 1. *When a user, after having identified the phone as a source object, identifies the internet radio as a sink, the display of the internet radio displays a message: "Alarm can be shared" and "Music can be played". Previews can also be less explicit, like briefly sounding an* *alarm and playing a short music clip. Note that the preview can be ignored or bypassed by establishing a connection.*

Example 2. *For exploring a connection between the internet radio and the dimmable lamp, the lamp simulates a wakeup sequence, increasing the light level from zero to its maximum intensity in a given period of time (in our implementation three seconds). This may be enhanced with simulating an alarm at the Squeezebox radio when the maximum of the intensity is reached.*

Practically, this means that the designer/developer of a smart object should design the response to a `PreviewEvent`. Technically, this is implemented by having the Connector object create a temporary connection to the devices to be connected in order to generate a `PreviewEvent`. This `tempConnectedTo` property is a sub-property of the `connectedTo` property (which denotes a regular semantic connection). This means that the smart objects will handle it as if it is a regular connection, and when the Connector object removes the `tempConnectedTo` relationship, the inferred `connectedTo` relationship will disappear as well. The type of functionality the preview is for, is added to the preview event as a data value.

The system behaves differently depending on the type of relation between the smart objects. When there is an indirect connection, i.e. going through a semantic transformer, the preview event is sent to the semantic transformer (Figure 31) instead of the sink object directly (Figure 30). Additionally, a temporary connection is made between the semantic transformer and the sink, ensuring that the sink displays the correct feedforward when the `PreviewEvent` is received.

5.4.1.2 *Functional feedback*

In many cases functional feedback of a semantic connection is trivial, for example hearing sound from a speaker that was just connected to a media player, or seeing photos on a TV when it is connected to a smart phone. However, functional feedback may only be available at another place or at another time. If we for instance take the example of synchronising a phone's alarm with the alarm radio, the real functional result may be hearing the radio play a song at the alarm time that was set on the phone.

In such cases, the interaction designers should use augmented feedback as an *indicator* that the alarm time was successfully set.

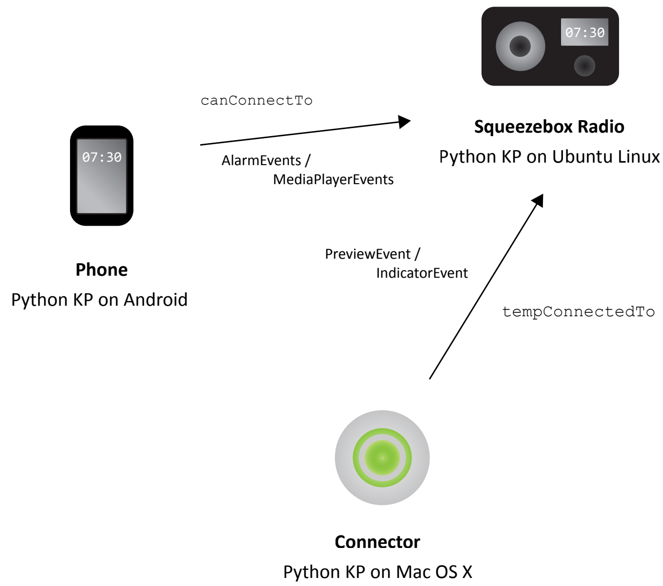


Figure 30: Temporary connections for a `PreviewEvent` when source and sink are directly connected

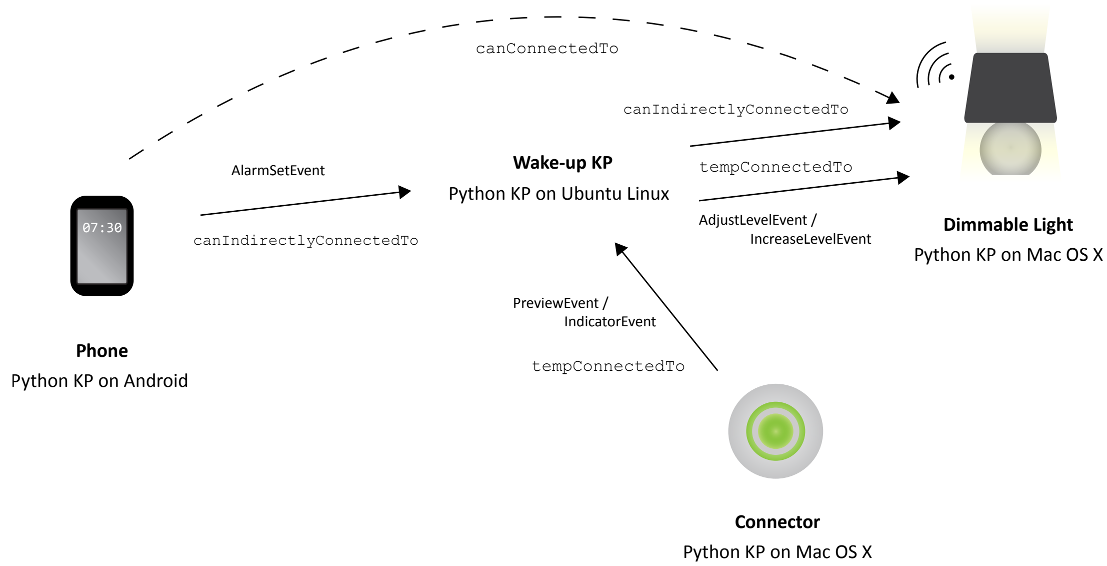


Figure 31: Temporary connections for a `PreviewEvent` when source and sink are connected via a semantic transformer

When a semantic connection exists between a source and a sink, actions at the source should also be indicated at the sink.

If the source and sink objects are in different locations, interaction designers should make sure that feedback is visible for a prolonged time period, or until it is dismissed by the user. This is to ensure that the indicator of the performed action will be noticed by the user. For the same reason, the order of connecting two spatially separated objects together is important, ensuring that establishing the connection happens in proximity of the sink, so that the feedback can be observed.

Example 3. *When music is playing on the phone and a connection is made between the phone and the internet radio, functional feedback is immediately given: the internet radio starts playing the same music, and an image of the album cover is displayed. The music on the source (phone) is muted, as the music playing on the internet radio is of a higher fidelity, and both share the same physical space. Context information, such as place/location, can be used to infer the correct behaviour.*

5.4.1.3 *Augmented feedback*

If there is no immediate link between action and function (e.g. functional result is delayed, information is given about an internal state change), augmented feedback can be used to provide this information. We use an `IndicatorEvent` to provide augmented feedback when smart object is connected and there is no immediate functional feedback, e.g. a sink "beeping" when the alarm is set on the source. The type of feedback required depends on the functionality of the connection. It is important for the feedback to coincide in time and modality with the event generated, as to maintain the causal link that is perceived by the user.

When a connections exists and an action performed on the source that has no immediate functional feedback, augmented feedback is provided to serve as an indicator. This feedback is provided by the smart objects that are connected, in the modality that is supported by their interaction capabilities. Designers should aim for maintaining the modality of the augmented information across the smart objects. Additionally, ensuring that the feedback occurring at distributed objects coincide in time may strengthen the perceived causality of the link. Indicator events may also be used to indicate existing connections, e.g.

when a user wishes to see what smart objects are currently connected to a source.

Example 4. *When the phone is connected to the internet radio and the internet radio is connected to the dimmable lamp, both the internet radio and the lamp give augmented feedback when an alarm is set. The internet radio displays the alarm-set screen, confirming the alarm time and the dimmable lamp slowly flashes, to indicate that they are both connected and that the action on the source is confirmed.*

5.5 DISCUSSION & CONCLUSION

For an evaluation of the design-related aspects of the third design iteration, please refer to [116].

The implementation of the sleep use case, described in Section 5.4, acted as an evaluation of the completeness and applicability of the concepts and techniques described thus far. The ontology and software framework that evolved during the previous two design iterations were applied to a new domain, testing the general applicability of the ontology's modelling capabilities as well as the software framework. These approaches were distilled into a theory of semantic connections, which forms the basis of the next chapter. The use case implementation evaluated:

- whether the concepts in the ontology are sufficiently defined to use them to implement the required functionalities;
- whether the defined concepts and techniques can be used universally (for different use cases); and
- whether the defined concepts and techniques form a complete set to describe the behaviour of semantic connections.

The outcome of the evaluation was favourable for all of the above. Additionally, the implemented use case can serve as an example of how the theory described in the next chapter is used in a relevant and contemporary setting.

We consider the length of the verbal descriptions in Section 5.4 above to become unwieldy when describing more complex situations. There is a clear need for some kind of diagram notation to describe these situations. In the next chapter we introduce a theory of semantic connections that was created to help solve this problem. We first describe the concepts that are central to this theory, and then introduce a diagram notation based on FSMs, that can be used to model and explain the different concepts and situations.

6 SEMANTIC CONNECTIONS FRAMEWORK

Essentially, all models are wrong, but some are useful.

— George Box [17], statistician

We refer to this as a theory and framework interchangeably, as we acknowledge that it is not a theory in the classical sense of the word, as used in the field of computer science.

Extracting from the lessons learned during the three design iterations, a theory was developed for interacting with a system of devices in a ubiquitous computing environment. This chapter introduces a theory of semantic connections, in which the connections and associations between devices play a central role. Semantic connections focus on the semantics—or meaning—of the connections between entities in a smart environment.

The theory may be used to analyse (i.e. understand, explain and predict) what happens with interaction events and other interaction data when devices are interconnected and form an ecology of smart objects. As an introduction to the Semantic Connections theory, we first focus on the Semantic Connections user interaction model.

6.1 USER INTERACTION MODEL

A user interaction model for semantic connections is shown in Figure 32. It describes the various concepts that are involved in the interaction in a smart environment and shows how these concepts work together. The interaction model was inspired by the MCRpd by Ullmer and Ishii [112] which in turn was based on the MVC model, both of which are described in Section 3.

The terminology of inherent, augmented and functional feedforward and feedback is adopted from [124] and was previously introduced in Section 2.5.1.

We first distinguish between the physical and digital domains of the user interaction. A user does not observe directly what is happening in the digital domain, but experiences the effect it has in the physical world by interacting with various smart objects. Semantic connections exist between these objects. By interacting with the objects, users create a mental model of the system that they are interacting with, which only partly includes the digital domain. The digital part manifests itself in the physical world as data, media and services.

When a user interacts with a smart object, he/she senses feedback and feedforward, directly from and inherent to the controls of the device (inherent feedback), digital information aug-

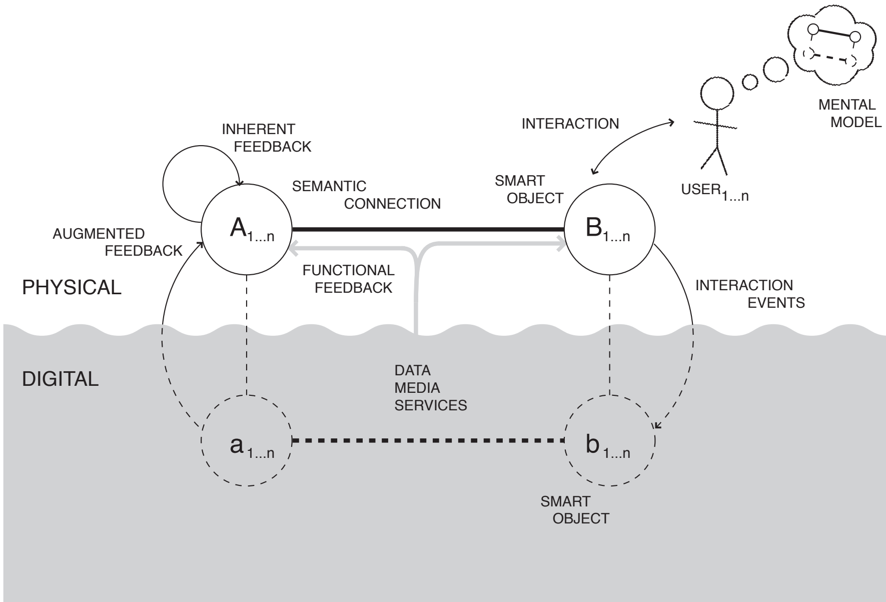


Figure 32: Semantic connections user interaction model

mented onto the physical world (augmented feedback) and perceives the functional effect of the interactions (functional feedback). The user actions in the physical world are transformed into interaction events and device state changes. This interaction data in terms of user intentions is stored in the smart space. The notion of a smart space means that data are stored either by an information broker or the smart objects themselves, and can be accessed by the other smart objects in the smart space. We use the term smart environment as a broader term to refer to both the digital and physical spaces, which include both the smart space and the smart objects.

Note that in Figure 32 the arrow on the left shows the feedback (or output) of the smart object, and on the right it shows the input (the interaction event). This to avoid repetition, as they may and in most cases will, occur on both sides.

6.2 SMART OBJECTS

Smart objects are the devices that are connected to the smart space, enabling them to share information with one another. We now define a smart object as follows:

Note that our definition does not define where the KP is situated. For simple smart objects, such as a smart light bulb, the actual KP that is communicating to the information broker may be a virtual entity running on any device in the network.

SMART OBJECT A smart object is a device with both computational and network communication capabilities that can be uniquely identified in both physical and digital space.

According to our definition, an NFC enabled smart phone is a smart object. A WiFi-connected lamp is also a smart object, given that it can be physically identified, for example by proximity based on signal strength or RFID.

In terms of our definition, a light switch with an RFID tag is not a smart object. A software agent running on a GUI (e.g. Microsoft Office's Clippy[1]), is not considered a smart object, even though it is visually perceivable. Despite its apparent physical existence, i.e. physically and digitally identifiable, it is mediated by a computer. In such cases the computer is considered the smart object and not the agent running on it, as the agent is not primarily a physical entity. In the following subsections we describe the concepts specific to the smart objects themselves.

6.2.1 *Identification*

For semantic connections to work in the way they are envisioned in this thesis, a smart object needs to be uniquely identifiable in both the physical and the digital domain. In the physical space it needs to be both user-identifiable and machine-identifiable. A device that is tagged with an RFID tag is machine-identifiable in the physical space, and the unique identifier read from this tag is also linked to the digital representation of the smart object. NFC—using a near field channel like RFID or infrared communication—is an interesting case, because it allows for direct manipulation of wireless network connections by means of proximal interactions [96].

Of course, there are many ways in which a smart object may be identified. An IP address makes a device easily identifiable in the digital space, but it is difficult to create a physical representation of this identity. Consider the case where IP addresses are printed on stickers and stuck on computers to make them easier identifiable to IT service personnel. One solution to making these stickers machine-identifiable again is to use Quick Response (QR) codes, two-dimensional barcodes which can be identified by mobile phone cameras.

6.2.2 *Interaction primitives*

We defined interaction primitives as a way to describe the user interaction capabilities of smart objects in ubiquitous computing environments. These interaction primitives are based on the work of Foley, Card and others introduced in Section 2.4.

Interaction primitives were introduced in Section 4.2.2.

The key on a keyboard labelled "A" is an interaction primitive, as pressing it not just changes the key's `Up` state into a `Down`

1 http://en.wikipedia.org/wiki/Office_Assistant

state, but carries the meaning to *produce* a character "A". A gesture `SwipeLeft` on a touchpad is also an interaction primitive, as this is the smallest addressable element to still have meaning. Describing the input on a lower level would cause it to lose its meaningful relation to the interaction. A touchpad itself is not an interaction primitive but rather an input device. An interaction with the touchpad, annotated with its meaning, can be an interaction primitive. A GUI is not an interaction primitive, but a GUI element can be.

Interaction primitives are described in terms of their physical properties that are meaningful to a user. For example, an unlabelled button should not only be represented in terms of its `On` and `Off` states, but also whether it is in a `Up` or `Down` state. This enables the mapping of physical, generic interaction primitives like a rocker switch to specific high-level events like `VolumeUpEvent`.

An interaction primitive also has a range measure that describes the range of possible values that it can take on. This makes it easier to determine if and how they can be mapped to specific interaction events.

Interaction primitives and interaction events together form an *interaction path* [36]. As an example, a typical interaction path would be:

`VolumeSliderLeft` → `SlideLeftEvent` → `VolumeDownEvent`

where the `VolumeSliderLeft` is an interaction primitive mapped to the `SlideLeftEvent` interaction event. Based on the available context information, this can in turn be mapped to a more specific `VolumeDownEvent`.

When modelling interaction primitives, only that which is meaningful to be shared with other devices is considered. It is not necessary to describe interactions that are internal to the device and that are not shared. An accelerometer, for example, may be modelled as a separate device, sharing the raw accelerometer data to be used by other devices. However, when integrated into a smart phone, the accelerometer's data can often be abstracted as part of an interaction path, e.g. to only share the orientation of the device, or specific gestures measured with the accelerometer. In this case, the raw values may only need to be available locally on the device, to be used by the developers of other device-specific applications.

Figure 33: Nokia Play 360° speaker system and N9 mobile phone

One of our academic partners in the SOFIA project, the University of Bologna, created an independent implementation of our interaction primitives [9].

6.3 SEMANTIC CONNECTIONS

Semantic connections is a term we introduced [117, 118] to describe meaningful connections and relationships between entities in an ecosystem of interconnected and interoperating smart objects.

Semantic connections were introduced in Chapter 3.

The connection between a remote control and a wirelessly controllable (on/of or dimmable) light bulb is a semantic connection. The connection exists between two smart objects that can be physically identified and connected through physical proximity. The connection's communication technology is unknown to its user and the remote control and light are conceptually linked by users, based on the perceived behaviour.

Another example of a meaningful connection is the Nokia Play 360° speaker system, shown in Figure 33, where music can be streamed wirelessly to the speaker using an NFC-enabled smart phone. By touching the phone to the top of the speaker, a connection is created that conceptually "carries" the music from the phone to the speaker.

The WiFi connection between a smart phone and a WiFi router is not a semantic connection, as the connection in itself has no clear meaning. A USB cable by itself is also not considered a semantic connection.

Semantic connections make up a structural layer of inter-entity relationships on top of the network architecture. These connections can be the real, physical connections (e.g. wired or wireless connections that exist between devices), or conceptual connections that seem to be there from a user's perspective. Semantic connections exist in both the physical world and the digital domain. They have informative properties which are perceivable in the physical world. However, some of these physical qualities might be hidden by default, and only become visible on demand by means of a mediating interaction device. The digital parts of semantic connections are modelled in an ontology. There may be very direct mappings, e.g. a connection between two real-world entities may be modelled by a `connectedTo` relationship between the representations of these entities in an ontology. Sometimes the mapping is not so direct, for example where a semantic transformer is used. Semantic connections have several properties, which are explained in the following subsections.

6.3.1 *Directionality*

As discussed in Section 5.2, we consider a semantic connection to have a specified direction, or to be bidirectional/symmetric. Smart objects that are connected should then be identified as sources and/or sinks. Directionality may intentionally be specified through user action, or it can emerge from the capabilities of the smart objects e.g. connecting a source to a sink will automatically create a connection going from the source to the sink.

6.3.2 *Transitivity*

When connections have directionality and multiple devices (i.e. a minimum of three devices) are involved, devices can also act as bridges, transferring data due to transitivity. For example, if a music player is connected to speaker A, and speaker A is connected to speaker B, speaker A acts as a bridge between the music player and speaker B.

6.3.3 *Permanent and temporary connections*

Semantic connections can vary in persistence. Connections can be made during an interaction cycle involving several devices to transfer content or data from the one device to another, and the connection then stops existing when the interaction cycle is completed. Connections can also be used to configure more permanent information exchange between entities in a smart space, much like setting up a connection to a wireless network router. These permanent connections will persist, and will be automatically reconnected every time the smart objects that are connected co-exist in the same smart space.

6.3.4 *Connections connect different entities*

Connections can exist between smart objects, people and places. Not only objects and devices have meaning in a system of networked devices — according to [92] physical location within the home and device ownership (or usage) are of central importance for understanding and describing home networks by users. Ownership can be seen as a connection between a device and a person. Connections from and to places or locations can be seen as a way of structuring contextual information such as location. With very personal devices (such as smart phones and laptops or tablets) we can, when these devices are used in an interaction, implicitly infer the user's identity. With shared devices, we need a way to identify the user. In such cases, making explicit connections from the device at hand to something personal of the user (e.g. a phone or keychain) may be a way to indicate identity.

6.4 SEMANTIC TRANSFORMERS

Semantic transformers were introduced in Section 4.2.1.

Semantic transformers were first defined in [81] as virtual entities that transform one type of information into another when a direct mapping is not possible. They transform user actions into interaction events and perform matching and transformation of shared data and content. Semantic transformers enable interoperability between devices by utilising device capability descriptions and content types to determine how devices may interoperate.

Semantic transformers can be used to map and transformed shared content between smart devices, for example a service that transforms a music stream into coloured lighting patterns that can be rendered by a lighting device. Semantic transformers can also be used to transform physical actions (such as pressing a button or performing a gesture) into representational events like adjusting the level of lighting in a room, or the adjusting the volume of a speaker. Semantic transformers may also be employed to perform simpler transformations such as inverting values.

Physical identifiable objects are not considered semantic transformers and should rather be modelled as smart objects. Semantic transformers are services, and therefore have no physical appearance or tangible form. They can only be perceived through the smart objects they transform the information for. A semantic transformer is not considered a smart object, as it is a virtual object and not addressable in the physical environment.

6.5 FINITE STATE MACHINE EXAMPLES

We now use FSMs to model and explain the different concepts introduced so far. FSMs allow us to talk about user interaction in a way that describes how users could think about user interaction, but that still makes sense to interaction programmers and designers [109]. The use of FSMs also encourages simplicity.

States where the light bulb is removed or broken are not described, as these are discussed extensively in [109].

As a first example, consider a simple light with an up/down switch as a single device (seen in Figure 34 on the left). There are two states (`On/Off`), an initial state (`Off`) and two events (`SwitchDownEvent/SwitchUpEvent`) that cause transitions between the states. If the switch is labeled, we can use more specific (meaningful) wording, for example `switchOffEvent` instead of `switchDownEvent` (as shown in Figure 34 on the right).

In Figure 35 one of the simplest examples of a semantic connection is shown - a light (as a smart object) connected to a simple up/down switch (a second smart object). The light consists of two states (`On/Off`) with an initial (default) state of `Off`, and two events (`LightOnEvent / LightOffEvent`) indicating the transitions between these states. Boxes with rounded corners are used to signify smart objects, while the semantic connection is indicated using a solid arrow point. Using arrows to denote semantic connections allows us to specify a direction

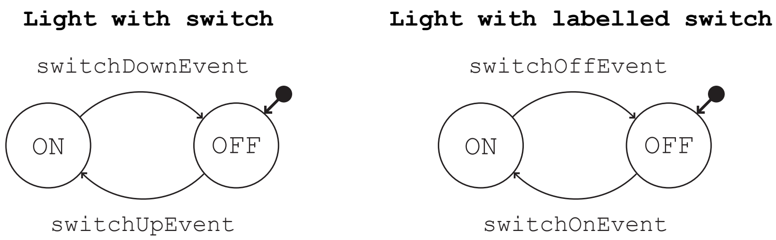


Figure 34: FSMs for a simple light with a switch and a light with a labelled switch

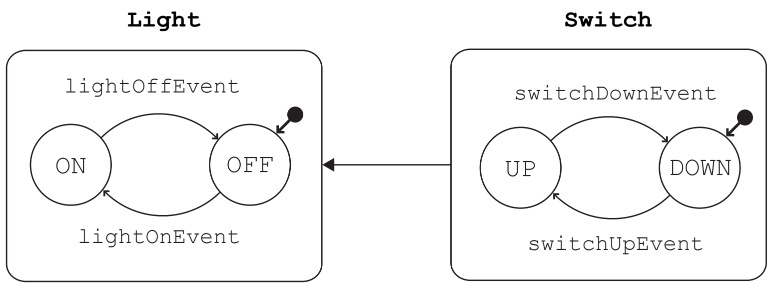


Figure 35: Light and light switch as two separate smart objects with a semantic connection

The concept of directionality was described in Section 6.3.1.

for the connection. Note that the light has functional feedback, with perceivable light when it is switched on. The switch on the other hand has inherent feedback, with a perceivable `Up` or `Down` state.

We can create mappings between the events to create an interaction path (see section 6.2.2), for example we use

$$\text{SwitchUpEvent} \rightarrow \text{LightOnEvent}$$

to indicate the most meaningful *default* mapping. It should of course be possible to change this mapping, for example by using a semantic transformer that inverts mappings between devices.

In the case where a smart object is not in the same physical location as the smart object it is connected to, additional augmented feedback may be required. Consider the case where the light switch may be in a different room than the light - we could use an indicator on the switch to give augmented feedback to show whether the light actually switched on. This is shown in Figure 36.

A more complex example is shown in Figure 37. In this example there is a symmetric (bidirectional) connection between

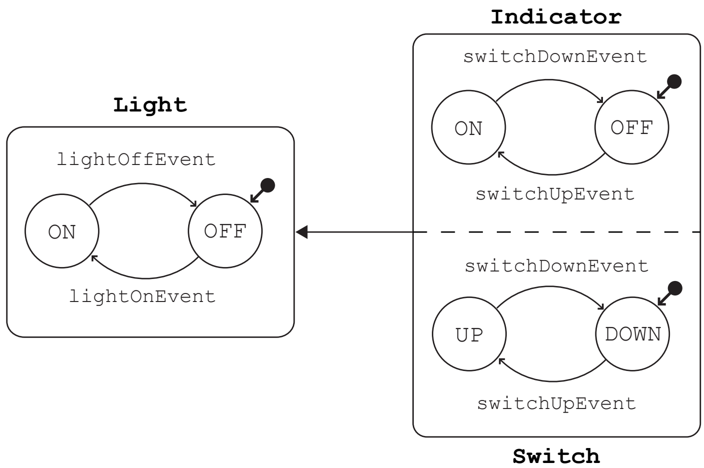


Figure 36: Light connected to light switch with augmented feedback

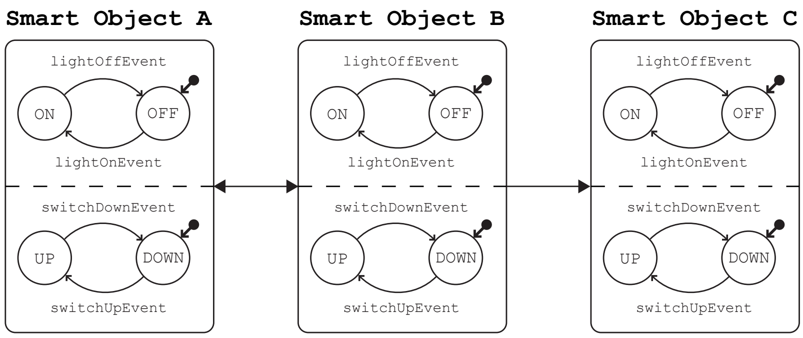


Figure 37: FSM showing semantic connection with symmetry

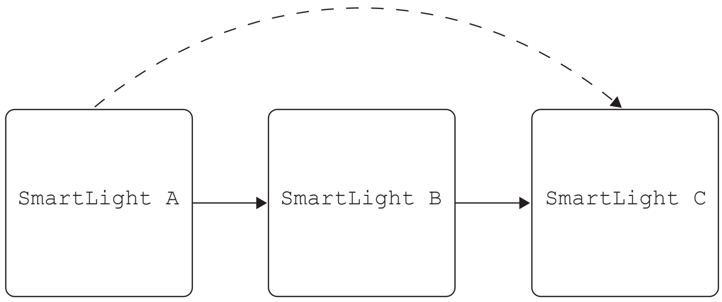


Figure 38: FSM showing a semantic connection with transitivity

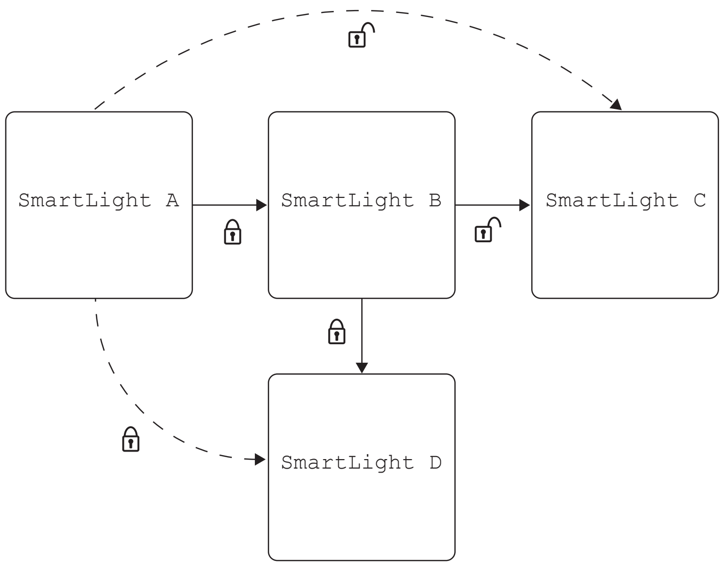


Figure 39: A semantic connection with transitivity and persistence

smart object A and B, with the result that pressing the switch on smart object A will turn the light in smart object B either on or off, and vice versa, B will control A. Since B is connected to C, actions on A and B, will also be reflected on C. On the other hand, pressing the switch of C will have no effect on either A or B. Due to the symmetric connection, we expect A and B to be in an identical state.

One possible solution for such a light switch is a push button with an indicator light, such that the switch able to change its state by itself.

In Figure 38 we use the SmartLight abstraction to denote the FSM of a smart light as shown in previous figures. When SmartLight A is connected to SmartLight B, and in turn is connected to C, transitivity allows us to infer a direct connection (indicated by a dashed arrow) between A and C. Pressing the light switch on A will in this case affect both B and C.

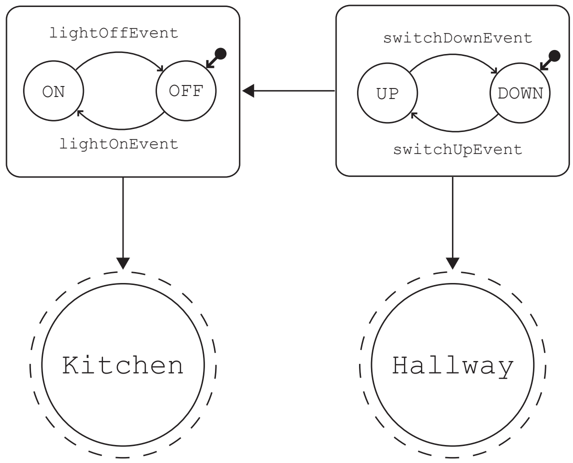


Figure 40: Semantic connections between smart objects and places

We use locked/unlocked icons next to semantic connections to indicate persistence (see Figure 39). The locked icons between A, B and D indicate persistent connections between those objects, and a persistent transitive connection is then inferred between A and D. This means that if smart object D moves to another location, all three connections (including the A→D connection) continue to exist. The connection between B and C is temporary, which means that the inferred transitive connection between A and C is also temporary. If smart object C moves to another location, both the B→C and A→C connections will be removed.

In Figure 40 we show semantic connections between smart objects and specific locations, where the dashed-double circle denotes a location. This places semantic connections between places and objects on the same abstraction level. We use semantic connections between smart objects and places as a way to structure relevant contextual information. In our example in Figure 40 we cannot infer that a user actually is able to observe the functional feedback of switching the light on and off, as they are not located in the same space, and might not be able to see the light. The importance of feedback and feedforward and how they should be handled between different locations is described in more detail in section 5.4.1.

When two switches are connected to the same light as is shown in Figure 41, the issue of priority arises. We define the most meaningful default to be that the last event that occurred

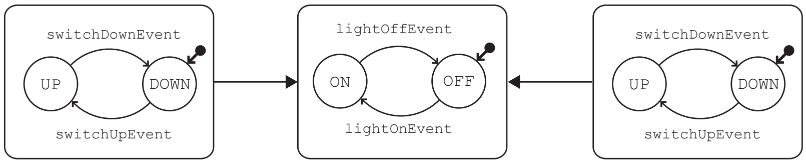


Figure 41: FSM showing a situation where the issue of priority arises

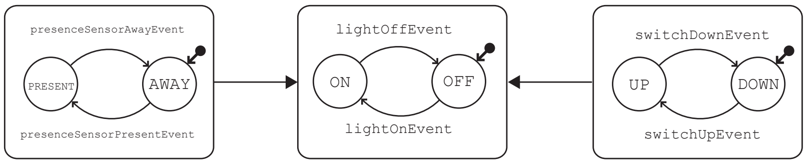


Figure 42: FSM showing incidental (presence sensor) and intentional (light switch) interactions

has priority. In Figure 42 where the one interaction is incidental, generated by a presence sensor, and the other is intentional (as described in Section 2.5.2), the intentional interaction takes priority.

6.6 FEEDBACK AND FEEDFORWARD

If we view our concept of semantic connections in terms of the Interaction Frogger framework (as discussed in Section 2.5.1), the following interesting insights emerge.

6.6.1 *Feedback of objects*

When we consider multiple interconnected smart objects and the functionalities and services they provide, information like feedback and feedforward gets spatially distributed. A user may operate a device, receiving inherent feedback locally, but receiving augmented and/or functional feedback remotely.

As inherent feedback is inherent to the operational controls of the device, these reside only in the physical world and are local to the device. We thus do not model this feedback in the digital domain. Augmented feedback is feedback that is augmented from the digital domain onto the physical world. This type of feedback is subject to change when devices are connected to other devices. In the domain of networked digital artefacts, functional feedback is of a digital nature. Data, media and services that exist in the digital domain become available in the physical world, through the various devices and their connections. In Figure 32, the several types of feedback are indicated.

Although many functionalities of digital devices can be regarded as displaying media, data or services, for some simple functionalities this seems problematic. If we, for example, look at functional lighting, it seems that the presence of light as the functionality of a lighting device is not a concept that is part of the digital domain. However, if we view a lighting device as a networked smart object, the presence of lighting, based on some sensor data, can be regarded as the functionality of a digital service.

Refer to Van der Vlist's thesis [116] for more detail on how the theory of product semantics can be applied to feedback and feedforward.

6.6.2 *Feedback of connections*

Inherent feedback becomes feedback that is mediated through an interaction device used to make or break the connection, as one can not manipulate a wireless connection directly. This inherent feedback may however be closely related to the action of making or breaking a physical connection, like a snap or click when the connection is made or broken. Augmented feedback to indicate a connection possibility or an existing connection may be in the form of lights, or in the form of projected or displayed lines. Functional feedback is information about the actual function of the connection, like music playback from a speaker that was just connected to a media player. This type of feedback always reaches the user through the devices being connected.

6.6.3 *Feedforward*

Inherent feedforward, conceptually similar to the notion of affordances [84], provides information about the action possibilities with the devices or the individual controls of an interface. Inherent feedforward is always physical and local on the device. However, when devices or objects are part of a larger system, feedforward also emerges where interaction possibilities between objects exist (e.g. a key that fits a lock, a connector of one device or cable that fits another). The same holds for augmented feedforward, where lights, icons, symbols and labels provide additional information about the action possibilities. These may concern the action possibilities locally at the device, as well as action possibilities that concern the interaction with other devices in the environment.

While inherent and augmented information are primarily concerned with "the how", functional feedforward communicates "the what", the general function of the device or the function of a control. This type of information often relies on association, metaphors and the sign function of products, and are described in theories such as product semantics and product language. With multifunctional devices, and even more with smart objects, this becomes increasingly difficult. Introducing the concept of semantic connections tries to address these problems, therefore the functional feedforward is the main challenge when designing semantic connections. Functional feedforward gives information about the function of the semantic connection before the interaction takes place. Properly designing functional feedforward is therefore the crucial part of understanding semantic connections, smart services and smart environments.

An example of where functional feedforward was used in the third design iteration is described in Section 5.4.1.

Wensveen [124] further proposes that in interaction, these types of information can link action and function together in time, location, direction, modality, dynamics and expression. Strengthening these couplings between action and function will lead to richer and more intuitive interactions [123].

We can also view semantic connections in the Frogger framework in more general terms. Although semantic connections are not a physical device or product, but rather describe the structure or configuration of a system of devices, the Frogger framework can teach us important lessons. When we look at the link between action and functional information in time or location, a strong link would mean they coincide in time and location. For location this would mean that the connection that is made between devices corresponds to the location of the actual devices in physical space. But also that the feedback that is provided is coupled to the action in time an location. Additionally, the direction of the action of connecting/disconnecting devices, being moving devices towards or away from each other, strengthens the coupling in terms of direction. Also, the direction of the action could have a link to the directionality of the semantic connection that is made (e.g. the order in which endpoints of a connections are defined). Couplings in dynamics (of the action) can be used in similar ways and may express the persistence of the connection that is made.

6.7 DISCUSSION & CONCLUSION

In this chapter we introduced our Semantic Connections theory and used finite state machines to model and explain the different concepts. We defined the following main concepts:

- Smart objects, and the means to describe them in terms of a unique physical and digital identity
- Interaction primitives, and how they can be used to describe the user interaction capabilities of smart objects
- Semantic connections, and how they can be used to model meaningful connections between smart objects
- Semantic transformers, and how they transform information from one type into another

We identified some of the principles of semantic connections, including directionality and transitivity, as well as permanent and temporary connections. We also identified principles of the various types of feedback and feedforward that are required, not only for connections but also for smart objects.

The importance of being able to uniquely identify smart objects in both the physical and digital space, as well as sharing their interaction capabilities and states, was shown, including how it was grounded in the theory of interaction models by Nielsen, Card and others.

We showed how augmented and functional feedback and feedforward can help users to better predict the functional result of the connections they create. Functional and augmented feedback also showed to be key in maintaining the causal links between user action and function, distributed over interconnected smart objects.

A fundamental difficulty encountered during the implementation of the feedback and feedforward (and which is also a big challenge in interoperability in general), is what we call the *awareness paradox*. To foster emergent functionality, efforts are aimed at enabling smart objects to interoperate without their combined functionality being specifically designed. This means that the smart objects are unaware of each other, exchanging information though an information broker. For the users however, it is imperative that smart objects show behaviour as to appear to be aware of each another.

The way out of the paradox is to make use of proper use of feedback and feedforward that can be generated at runtime. Since the connections that may be created during use are not known at design time, smart decisions have to be made on how to describe the interaction events and functionalities that are shared.

By describing feedback and feedforward of the semantic connections as a result of the match in capabilities and functionalities, and having the semantic transformers and sink objects (instead of the source) produce the preview and indicator feedback, we make sure that they are capable of displaying (i.e. in the widest sense of the word, not limited to the visual modality) this feedback. Our reasoning is that, if a sink can be the sink of a functionality, it should also be capable of giving feedforward and feedback for this functionality.

Examples of preview events were shown in Chapter 5.

Moreover we showed that semantic connections and using semantic transformers to create services is an appealing idea, leading to additional and, more importantly, more meaningful functionalities of ensembles of existing devices. This may reduce the number of devices needed in our daily lives by reducing redundant devices.

The theory of semantic connections described in this chapter provides a foundation for modelling user interactions with interoperating smart objects in smart environments, and therewith the possibility to improve the interoperability among them. We considered the notions of feedback and feedforward to enhance perception of connectivity and the perceived causality between user action and feedback.

In the next part of the thesis, we will look at other concepts and techniques that can be used in smart environments, like device capability modelling and event modelling. Similarly to how we extracted the theory of semantic connections from the work completed in the design iterations, these more general concept and techniques are also based on the work done during the three design iterations.

Part III

GENERALISED MODELS, SOFTWARE ARCHITECTURE AND EVALUATION

In this part of the thesis, the more general concepts and techniques that can be applied to ubiquitous computing are described, followed by the final iteration of the software architecture. These concepts and techniques were extracted from work done during the three design iterations. A performance evaluation and usability study are described, followed by a discussion and conclusion of the work.

7 DEVICE CAPABILITY MODELLING

Whenever we capture the complexity of the real world in formal structures, whether language, social structures, or computer systems, we are creating discrete tokens for continuous and fluid phenomena. In doing so, we are bound to have difficulty. However, it is only in doing these things that we can come to understand, to have valid discourse, and to design.

— Alan Dix [30], HCI researcher

In order to share device capabilities with other devices in the environment, we require ways to describe these capabilities. While we have touched lightly on some of the techniques in the thesis so far, this chapter will focus in more detail on the current state-of-the-art, as well as how we extended these techniques to create a new way of modelling device capabilities using ontologies.

Most of the existing work on modelling interaction capabilities focuses on GUI based techniques.

7.1 GUI-BASED TECHNIQUES

A *universal user interface language* describes user interfaces that are rendered by mapping a description's device-independent interaction elements to a target platform's concrete interface objects [69]. This allows developers to create the user interface in an abstract language without targeting a specific device. Examples of interface languages include User Interface Markup Language (UIML), Extensible Interface Markup Language (XIML), Personal Universal Controller (PUC) and International Committee for Information Technology Standards Universal Remote Console (INCITS/V2 URC). These languages allow devices to determine the most suitable presentation based on a predefined set of abstract user interface components.

UIML maps interface elements to target UI objects using a styling section, resulting in one styling section per target device type. However, it does not include a vocabulary to describe more abstract widgets [130]. PUC describes device functions in terms of state variables and commands, with a grouping mechanism used for placement of UI objects. The INCITS/V2 URC standards define a generic framework and an XML-based user interface language to let a wide variety of devices act as a remote to control other devices, called targets.

User interface remoting uses a remote interface protocol that relays I/O events between an application and its user interface. The user interface resides on a remote platform instead of on the device itself. The UPnP Remote User Interface (RUI), that forms part of the UPnP AV standard, belong to this category. UPnP RUI follows the Web server-client model, where the controller acts as a remote user interface client, and the target, acting as a remote user interface server, exposes a set of user interfaces [113].

CEA-2014, that builds on the UPnP RUI interface, uses a matchmaking process for a controller device to select a user interface protocol that is supported by the controller platform [130]. Supported protocols include AT&T Virtual Network Computing (VNC) and Microsoft Remote Desktop Protocol (RDP). VNC uses the Remote Framebuffer (RFB) protocol to send pixels and event messages between devices.

Universal user interface languages and user interface remoting are orthogonal approaches [69]. User interface remoting might be used in parallel with device-independent user interface languages.

In this thesis we are more interested in tangible interactions in ubiquitous computing environments, instead of the usual GUI-based solutions. Smart environments need not only descriptions of GUI-based input/output, but also descriptions of the physical input/output capabilities, hardware capabilities, network capabilities and other characteristics of smart objects. The first attempt to define a vocabulary that conveys these device characteristics was the W3C Composite Capabilities/Preferences Profile (CC/PP)[1]. Other approaches to describe device characteristics that are not GUI specific are described in the next section.

1 http://www.w3.org/Mobile/CCPP/

7.2 NON-GUI TECHNIQUES

7.2.1 *UAProf*

The WAP Forum's User Agent Profile (UAProf) specification is an RDF-based schema for representing information about device capabilities. UAProf is used to describe the capabilities of mobile devices, and distinguishes between hardware and software components for devices.

W3C's CC/PP is also an RDF-based schema.

For example, in the Nokia 5800 XpressMusic UAProf profile[2], its interaction capabilities are described as follows:

- `PhoneKeyPad` as `Keyboard`
- 2 as `NumberOfSoftKeys`
- 18 as `BitsPerPixel`
- 360x640 as `ScreenSize`
- Stereo as `AudioChannel`

Other user interaction capabilities are defined in a Boolean fashion of yes/no, e.g. `SoundOutputCapable`, `TextInputCapable`, `VoiceInputCapable`.

7.2.2 *Universal Plug and Play (UPnP)*

UPnP with its device control protocols is one of the more successful solutions[3] to describing device capabilities. However, it only allows for the definition of one level of tasks [80].

UPnP was developed to support addressing, discovery, eventing and presentation between devices in a home network, and the current version (1.1) was released as the ISO/IEC 29341 standard in 2008. It consists of a number of standardised Device Control Protocols (DCPs) - data models that describe certain types of devices. The DCPs that have been adopted by industry include audio/video, networking and printers. DCPs for low power and home automation have not yet been adopted.

Digital Living Network Alliance (DLNA) is a complete protocol set around Internet Protocol (IP) and UPnP, where to be certified for DLNA, a device needs to have UPnP certification

2 nds1.nds.nokia.com/uaprof/Nokia5800d-1r100-2G.xml
3 http://upnp.org/sdcps-and-certification/standards/sdcps/

first. This protocol set was developed mainly to increase interoperability between Audio/Video (AV) equipment in the home. It achieves this by limiting the amount of options available in the original protocol standards.

In an IEEE ComSoc online tutorial entitled Consumer Networking Standardizations, *Frank den Hartog from TNO stated that "DLNA has been a major effort to get computer people to talk to consumer electronics people".*

When describing the capabilities of a smart object, not only the interaction capabilities are important, but also the device states. With UPnP, two types of documents are used to describe device capabilities and states. A *device description document* describes the static properties of the device, such as the manufacturer and serial number [62]. UPnP describes the services that a device provides in *service description documents*. These XML-based documents specify the supported actions (remote function calls) for the service and the state variables contained in the service.

The state variable descriptions are defined in a similar way to how we define our interaction primitives, with a unique name, required data type, optional default value and recommended allowed value range. The UPnP Forum has defined their own custom set of data types, with some similarity to the XML Schema data types used by OWL 2. As an example, consider a state variable to describe the darkness of a piece of toast, where `ui1` is defined as an unsigned 1-byte integer:

```
<stateVariable sendEvents="no">
        <name>darkness</name>
        <dataType>ui1</dataType>
        <defaultValue>3</defaultValue>
        <allowedValueRange>
                <minimum>1</minimum>
                <maximum>5</maximum>
                <step>1</step>
        </allowedValueRange>
</stateVariable>
```

The `sendEvents` attribute is required for all state variable descriptions. If set to `"yes"`, the service sends events when it changes value. Event notifications are sent in the body of an HTTP message and contain the names and values of the state variables in XML.

Let us consider these device states in terms of user interaction. There are four key concepts in an interaction - actions, states (internal to the device), indicators, and modes (physically perceivable device states) [109]. The user performs actions, which change the device state, which in turn control indicators (augmented feedback). Users may not know exactly which state or mode a system is in. If we want to fully capture the capabilities of the device, we need to specify the device states, the transitions between these states, the interaction primitives which can cause these state changes, as well as the default and current states of the device. When this device is then connected to another device, we also need a way to communicate state changes to the other device.

Our approach to modelling devices states and state transitions using FSMs is described in Section 6.5.

7.2.3 *SPICE DCS*

The Service Platform for Innovative Communication Environment (SPICE) Mobile Ontology[4] allows for the definition of device capabilities in a sub-ontology called Distributed Communication Sphere (DCS) [121]. A distinction is made between device capabilities, modality capabilities and network capabilities. While the ontology provides for a detailed description of the different modality capabilities, e.g. being able to describe force feedback as a `TactileOutputModalityCapability`, there are no subclass assertions made for other device capabilities. Most physical characteristics of the devices are described via their modality capabilities, e.g. a `screenHeight` data property extends the `VisualModalityCapability` with an integer value, and the `audioChannels` data property is also related to an integer value with `AcousticModalityCapability`. The input format of audio content is described via the `AcousticInputModalityCapability` through an `inputFormat` data property to a string value.

7.3 REGISTERING DEVICES ON STARTUP

Based on this existing work, we now look at our approach to registering device functionalities, as well as how we identify devices in the digital and physical domain.

On device startup, the smart object registers its digital and physical identification information (e.g. RFID tag or IP address) and its functionality with the SIB, and then subscribes to new connections and events as shown in Figure 43.

The startup sequence contains instances of the blackboard and publish/subscribe patterns described in Section 10.2.

This sequence is the same for all smart objects that connect to the SIB, and should be implemented in every KP that uses our

4 http://ontology.ist-spice.org/

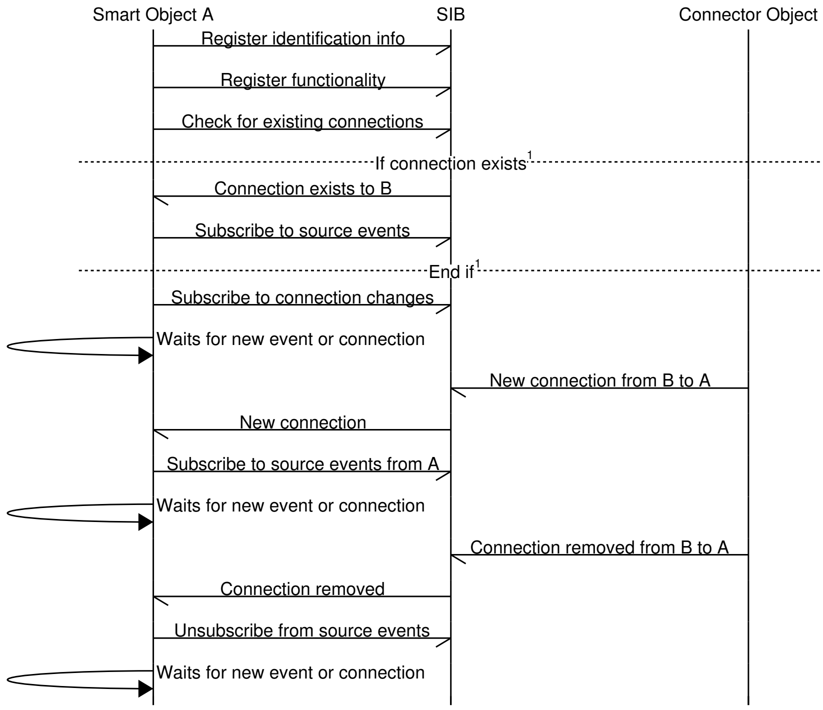


Figure 43: Startup sequence between smart object and SIB

approach. You might notice some parallels between the concept of a SIB and the Microsoft Windows Registry. The Registry is used to store configuration information of software applications on a single device, while the SIB is used (among other things) to store device functionality descriptions of a system of devices. However, compared to the Windows Registry, which is a basic hierarchical key-value store, the triple store and reasoning engine used in the SIB provide a number of advantages, including subsumption testing, consistency checking and the ability to use restrictions to constrain data instances. This means we can use reasoning to verify the consistency and stability of the data in the SIB.

Subsumption testing, consistency checking and restrictions are discussed in more detail in Section 9.3.

We now discuss the first two steps of the sequence diagram in Figure 43, registering identification and functionality, in more detail.

7.3.1 *Identifying devices*

In order to discover a device's capabilities, it is first necessary to be able to uniquely identify the device. Today it is common to identify groups of products using barcodes and other numbering systems. Before ubiquitous computing, only expensive things such as precious metals, currency, or large machines were individually identified with any regularity [66]. New tracking technologies like RFID tags and smart cards allow us to link a unique identification number to a specific physical product, like a smart phone that identifies a specific person to the phone network. IPv6, an extension to the Internet Protocol standard, allows us to identify approximately 3.4×10^{38} objects in the digital domain.

Mavrommati et al [73] linked an XML-based description of an object's properties, services and capabilities with an artefact ID. This alphanumeric ID is mapped to a namespace-based identification scheme, using a similar process to the one used for computer MAC addresses.

Tungare et al [110] identified an information object in their Syncables framework, used to migrate task data and state information across platforms, via a Uniform Resource Identifier (URI). They used the structure

```
sync://<info-cluster-id>/<collection>/<type>/<path>/<object-name>
```

where the information cluster is the set of all devices a user interacts with during the course of a day. Each of the devices in an information cluster "offers a unique set of affordances in terms of processing capabilities, storage capacities, mobility constraints, user interface metaphors, and application formats".

Most service discovery mechanisms, for example those used by UPnP, assume the user will use the Internet to establish connections [62]. However, when we are in close proximity to things, we can address these things by pointing at them, touching them or by standing near them, instead of having to search or select them through a GUI.

Olsen et al. [89] used the domain name or IP address of a software client associated with a device to identify it, and a URL to identify services associated with a specific device. A user was associated with a URL used for that user's current session. The physical user was identified using a Java ring, with a small Java virtual machine running on the ring's microcontroller.

O'Reilly and Battelle [90] argue that formal systems for adding a priori meaning to digital data are actually less powerful than informal systems that extract that meaning by feature recognition. They think that we will get to an Internet of Things via a "hodgepodge of sensor data, contributing, bottom-up to machine-learning applications that gradually make more and more sense of the data that is handed to them". As an example, consider that using smart meter data to extract a device's unique energy signature, it is possible to determine the make and model of each major appliance.

Jeff Jonas's work on *identity resolution* uses algorithms that semantically reconcile identities [102]. His Non-Obvious Relationship Awareness (NORA) technology is a *semantically reconciled and relationship-aware directory* that is used by the Las Vegas gaming industry to identify cheating players within existing records. A semantically reconciled directory recognises when a newly reported entity references a previously observed entity.

We agree that waiting until every object has a unique identifier for the Internet of Things to work is futile. However, the Semantic Web was designed with this problem in mind. We can use a URI to identify an entity we are talking about. Different people will use different URIs to describe the same entity. We cannot assume that just because two URIs are distinct, they refer to the same entity [3]. This feature of the Semantic Web is called, the *Non-unique Naming Assumption*. When using OWL, it is necessary to assert that individuals are unique using the `owl:allDifferent` or `owl:differentFrom` elements. Individuals can be inferred to be the same, or asserted using `owl:sameAs`. For OWL classes and properties, we can use `owl:equivalentClass` and `owl:equivalentProperty`.

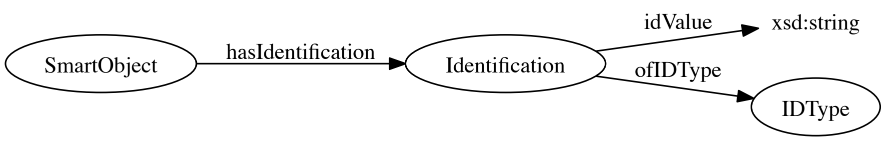


Figure 44: Modelling identification in the ontology

We modelled the identification of smart objects as shown in Figure 44. An example of how the Squeezebox KP can be linked to both its IP address and RFID tag is shown below:

```
SqueezeboxKP a SmartObject .
SqueezeboxKP hasIdentification id1234 .
SqueezeboxKP hasIdentification id4567 .
id1234 ofIDType IPAddress .
```

```
id1234 idValue "192.168.1.4:1234" .
id4567 ofIDType RFID_Mifare .
id4567 idValue "12AB45CD67EF" .
```

7.3.2 *Registering a device's functionality*

In the first design iteration we used a very simple approach to modelling the capabilities of devices, where `provides` and `consumes` properties linked smart objects to the names of the capabilities. During the later design iterations we modelled capabilities as functionalities of a device instead.

Examples of `provides` *and* `consumes` *were shown in Section 3.2.*

To register the functionality of a device such as the Squeezebox internet radio, we can use the following triples:

```
squeezeboxKP a SmartObject .
squeezeboxKP functionalitySource Alarm .
squeezeboxKP functionalitySink Alarm .
squeezeboxKP functionalitySink Music .
```

This indicates that the device is capable of acting both as a source and as a sink for `Alarm` functionality, while it can act as a sink for `Music` functionality. Once these device capabilities are registered, we can use semantic reasoning to infer which devices can be connected to each other.

7.4 REASONING WITH DEVICE CAPABILITIES

A smart object can have one or more functionalities that can be shared with other smart objects. As shown in the previous section, we model a functionality as

```
ie:Alarm a ie:Functionality .
ie:phone1 a ie:SmartObject .
ie:phone1 ie:functionalitySource ie:Alarm .
```

or in the case of modelling the functionality of a sink we use

```
ie:Music a ie:Functionality .
ie:speaker1 a ie:SmartObject .
ie:speaker1 ie:functionalitySink ie:Music .
```

To infer that two devices can be connected based on functionality as shown in Figure 45, we use an OWL 2 property chain:

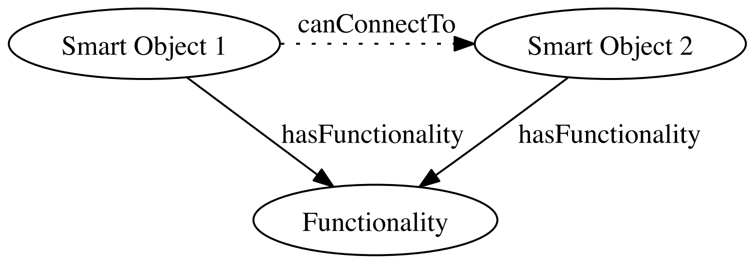


Figure 45: Inferring connection possibilities based on functionality

A similar method was used to match media types in Section 4.4.2.

$$\text{functionalitySource} \circ \text{isFunctionalityOfSink} \sqsubseteq \text{canConnectTo}$$

where we use `isFunctionalityOfSink`, the inverse property of `functionalitySink`, to be able to create the property chain.

To prevent a smart object from having a `canConnectTo` relationship to itself (which will be the case for semantic transformers), the relationship is defined to be irreflexive. Inferring indirect connection possibilities is also possible with a property chain:

$$\text{canConnectTo} \circ \text{canConnectTo} \sqsubseteq \text{canIndirectlyConnectTo}$$

7.4.1 *Representing functionalities as predicates*

Ontology design patterns are discussed in Chapter 9.

If we want to model the common functionalities between two smart objects, we can use the n-ary ontology design pattern [86]. Unfortunately, this is not intuitively readable from its ontological representation, as shown in the top half of Figure 46. The representation looks complicated and is difficult to read. On the other hand, we can also directly infer the matched functionalities as predicates, instead of using n-ary representations. The result can be represented using three triples instead of nine triples, and it is also more intuitively understandable, as shown in the bottom half of Figure 46.

Representing an individual as a predicate is not valid in OWL 2 DL and places the ontology into OWL 2 Full. However, since we are using an OWL 2 RL/RDF Rules reasoning mechanism, this is not an issue. Thus we choose to use predicates instead of n-ary relations, and so we do not stay within OWL 2 DL. We can easily infer this relation using a SPIN rule:

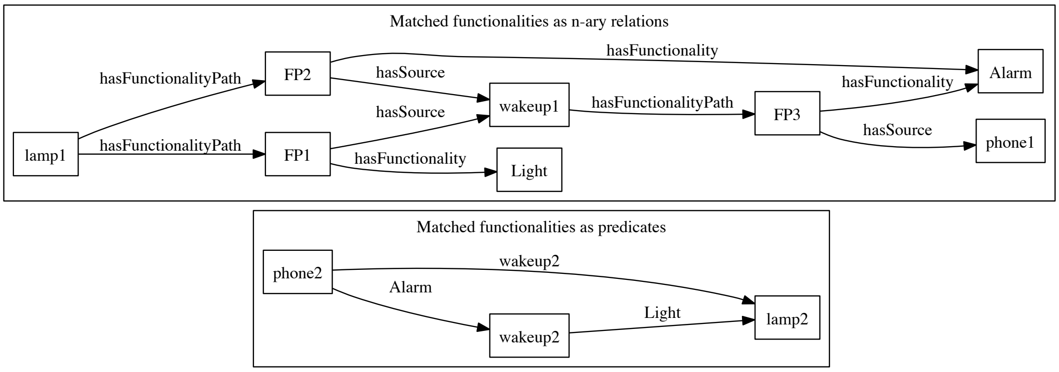


Figure 46: Representing matched functionalities: N-ary relations versus predicates

```
CONSTRUCT{
    ?this ?functionality ?sink .
}
WHERE{
    ?this :functionalitySource ?functionality .
    ?sink :functionalitySink ?functionality .
}
```

where `?this` ≡ `SmartObject`. For example, if we have a phone and a speaker with a common `Music` functionality, defined as

```
:phone1 :functionalitySource :Music .
:speaker1 :functionalitySink :Music .
```

the above SPIN rule will infer

```
:phone1 :Music :speaker1 .
```

such that the functionality itself is represented as a predicate. For a semantic transformer, which is indirectly connected to smart objects, we need an additional SPIN rule:

```
CONSTRUCT{
    ?source ?this ?sink .
}
WHERE{
    ?source :canIndirectlyConnectTo ?this .
    ?this :canIndirectlyConnectTo ?sink .
}
```

where `?this` $\equiv$ `SemanticTransformer`. This infers the semantic transformer itself as the relation between the source and the sink, since it transforms the original functionalities. For example, using the smart objects in Figure 46, if we have

```
:phone2 :functionalitySource :Alarm .
:wakeup2 :functionalitySink :Alarm .
:wakeup2 :functionalitySource :Light .
:lamp2 :functionalitySink :Light .
```

and using the two property chains from Section 7.4, we can infer that

```
:phone2 :canConnectTo :wakeup2 .
:wakeup2 :canConnectTo :lamp2 .
:phone2 :canIndirectlyConnectTo lamp2 .
```

If we then apply the SPIN rule defined above we can infer that

```
:phone2 :wakeup2 :lamp2 .
```

where the semantic transformer itself becomes the predicate between the two smart objects, signifying the possibility of having wakeup service functionality between the two objects.

How can we provide feedback or feedforward to the user that these possible functionalities exist between smart objects? This can be done using the feedback capability of the Connector object and the feedback capabilities of the devices themselves. Just after the user scans the second device, and before the connection is actually made, feedback of the different possibilities for shared functionality can be provided to the user.

The `dataValue` property of interaction events is discussed in more detail in Section 8.2.

Preview events and the `tempConnectedTo` property were first discussed in Section 5.4.1.1.

When two devices can be connected directly, the Connector object creates a temporary connection from itself to the sink. This temporary connection is specified using the `tempConnectedTo` property, a sub-property of the `connectedTo` property. The Connector object generates a `PreviewEvent` with the matching functionality as `dataValue`. This triggers the sink to create a preview of the functionality described by the `PreviewEvent` and its `dataValue`. When the sink completes the preview, it generates its own `PreviewEvent` to indicate that it has finished. The Connector object sees the sink's `PreviewEvent` and removes the temporary connection.

However, when there is a semantic transformer between the source and the sink, the Connector object creates a temporary connection to the semantic transformer instead of the sink, in

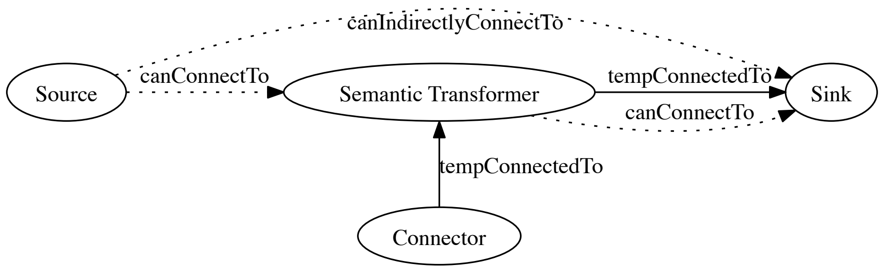


Figure 47: Temporary connections for `PreviewEvent` when semantic transformer is used

order to generate the appropriate `PreviewEvent`, as shown in Figure 47. Keep in mind that the semantic transformer is a virtual object, and therefore only the preview functionality generated by the sink will be perceivable by the user.

The Connector object uses the inferred `canIndirectlyConnectTo` and `canConnectTo` properties to determine where to insert the `tempConnectedTo` properties. After inserting the `tempConnectedTo` properties, the Connector object generates a `PreviewEvent`. Due to the `tempConnectedTo` relationship between the Connector object and the semantic transformer, the semantic transformer responds to this event and also generates a `PreviewEvent` with its own functionality as `dataValue`. Due to the `tempConnectedTo` relationship between the semantic transformer and the sink, the sink responds to the preview event of the semantic transformer and generates the appropriate preview of its functionality.

Now that we have a way to model the capabilities of devices and provide previews of their functionality, we can start looking at ways to represent the different kinds of events that are generated on these devices in the next chapter.

8 EVENT MODELLING

[Artefacts] mediate activity that connects a person not only with the world of objects, but also with other people. This means that a person's activity assimilates the experience of humanity.

— Aleksei N. Leontiev [60], founder of activity theory

Parts of this chapter appear in [81].

Events are notable occurrences that can be associated with people, places and objects at a specific time instant, or during a specific time interval. In the previous chapter the modelling of objects and their capabilities was described. The focus of this chapter is on how the event and its associated time instant/interval can be modelled. The modelling of people and places is considered to be outside the scope of this thesis.

Interaction events were first introduced in Section 3.2.

The W3C Web Events Working Group defined four conceptual layers for interactions, in the context of touch- and pen-tablet interaction [1]:

PHYSICAL This is the lowest layer, and deals with the physical actions that a user takes when interacting with a device, such as pressing a physical button.

GESTURAL This is layer describes mappings between the lower and upper layers; for example, a "pinch" gesture may represent the user placing two fingers on a screen and moving them together at the physical layer. This may map to a "zoom-in" event at the representational layer.

REPRESENTATIONAL This layer indicates the means by which the user is performing a task, such as zooming in, panning, navigating to the next page, activating a control, etc.

INTENTIONAL This layer indicates the intention of the task a user is trying to perform, such as viewing more or less detail (higher-level abstraction of zooming in and out), viewing another part of the larger picture (higher-level abstraction of panning), and so forth.

Interaction events can be defined at three of the layers, with the exception of the intentional layer. In Table 3 examples of possible interaction events are shown, together with possible

INTERACTION EVENT	CAN BE PERFORMED ON
AdjustLevelEvent	Volume, Lighting
switchOnEvent	Lighting, any SmartObject
NavigateEvent	Playlist, Menu, SequentialData
UndoEvent	Any other interaction event
StopEvent	Application, Media

Table 3: Examples of interaction events in a smart environment

entities associated with these events. Most of these interaction events exist at the representational layer, which are events that have significant meaning. This high level of abstraction enables developers to write applications which will work across different devices and services, without having to write specific code for each possible input device.

These events all occur at a specific time or during a specific time interval. In OWL, there are are two approaches to modelling time:

- Using datatype properties – event instances can be related to a literal with a XSD datatype such as `xsd:date` or `xsd:dateTime`.
- Using object properties – classes are used to define temporal intervals, and event instances are linked to instances of these classes using object properties.

The advantage of linking event instances directly with dates is simplicity. There are fewer abstractions to deal with, and it is easier to sort events chronologically and compare them [107]. On the other hand, working with temporal intervals provide more flexibility and allows for more detailed temporal reasoning.

This chapter is structured as follows: First we look at current approaches to modelling events using ontologies, which is then followed by our approach to modelling interaction events, based on the concepts of the existing event ontologies.

8.1 RELATED WORK

We now look at various existing event ontologies that we build upon to model interaction events in ubiquitous computing environments. We will also look at how temporal reasoning is performed with ontologies.

8.1.1 *The Event Ontology*

The Event Ontology (EO)[1] was developed within the context of the Music ontology[2] at Queen Mary, University of London. Although originally created to describe musical performances and events, it is currently the most commonly used event ontology in the Linked Data community [107].

The Timeline[3] ontology, used to define time instants and intervals, also forms part of this collection of ontologies. Reasoning with temporal information is discussed further in Section 8.1.5.

8.1.2 *DUL*

The DOLCE+DnS UltraLight (DUL) upper ontology is a lightweight version of the DOLCE ontology. DUL defines the class `Event` next to the disjoint upper classes `Object`, `Abstract` and `Quality` [101]. DUL allows for both the approaches to modelling time with OWL, either with the `hasEventDate` datatype property, or with a `TimeInterval` class and the `isObservableAt` object property.

Events can be related to a `Place` with the `hasLocation` property. Alternatively, events can be related to a `SpaceRegion` with the `hasRegion` property, where `SpaceRegion` resolves to a geospatial coordinate system. DUL uses a `hasParticipant` property to relate an event to an object, and uses the `hasPart` property to link events to sub-events.

8.1.3 *Event-Model-F*

The Event-Model-F ontology extends the DUL ontology to describe events in more detail. To describe the participation of an object in an event, the Event-Model-F ontology uses the DnS ontology design pattern [107].

The DnS pattern is described in more detail in Chapter 9.

An object is defined as a `Participant`, where `LocationParameter` is used to describe the general spatial region of the ob-

1 http://motools.sf.net/event/event.html
2 http://purl.org/ontology/mo/
3 http://motools.sf.net/timeline/timeline.html

ject [101]. `TimeParameter` describes the general temporal region when the event happened by parametrizing a DUL `TimeInterval`. A composite event `Composite` is composed out of a number of `Components`.

8.1.4 *Linked Open Descriptions of Events (LODE)*

The Linked Open Descriptions of Events (LODE) ontology[4] is an ontology for publishing descriptions of historical events as Linked Data. It builds upon the work of the previous ontologies described in this section, in order to improve interoperability with legacy event collections. Its `Event` class is directly equivalent to those defined by EO and DUL.

It uses time intervals to link events to ranges of time, where its `atTime` property is a sub-property of the DUL `isObservableAt` property. There is also a distinction between places and spaces, where the `inSpace` property relates the event to a space, and the `atPlace` property is a sub-property of the DUL `hasLocation` property.

8.1.5 *Ontologies for temporal reasoning*

Temporal reasoning is used when working with time intervals, for example when using Allen's Interval Algebra to define temporal relations between events. There are 13 base relations in this algebra, for example to define that event X happens before event Y, or that event X occurs during event Y. Allen's Interval Algebra is used by a number of ontologies, including the DOLCE [101] upper ontology.

The DOLCE upper ontology is discussed in more detail in Chapter 9.

SWRL Basic Temporal Built-ins support `xsd:date` and `xsd:dateTime` with Allen's Interval Algebra. The Advanced Temporal Built-ins uses the `temporal` ontology [87] to provide additional functionality, for example having different granularity levels.

The Dublin Core (DC) Terms ontology has a `temporal` property to describe temporal coverage of a resource with a range `periodOfTime`.

TopBraid Composer has a Calendar ontology that defines an `Event` and its `startTime` and `endTime` (as `xsd:dateTime`). This can then be used with the Calendar View widget in the editor.

4 http://linkedevents.org/ontology/

In SPARQL, we can use the < and > operators on dates, for example

```
FILTER(?date > "2005-10-20T15:31:30"^^xsd:dateTime)
```

Using SPIN, can also cast a `xsd:dateTime` value to a string using the `fn:substring` function, for example

```
fn:substring(xsd:string(afn:now()),0,10)
```

We build on these event models to create our own interaction event model. In the next section, a mapping between the entities in the various ontologies show how our work relates to the existing work.

8.2 interaction events

We now turn our focus to how interaction events are modelled in the work described in this thesis. In order to perform semantic reasoning with events, we need to know when they occur and be able to uniquely identify them. An interaction event happens at a specific time, is generated by a smart object and has an optional data value associated with the event.

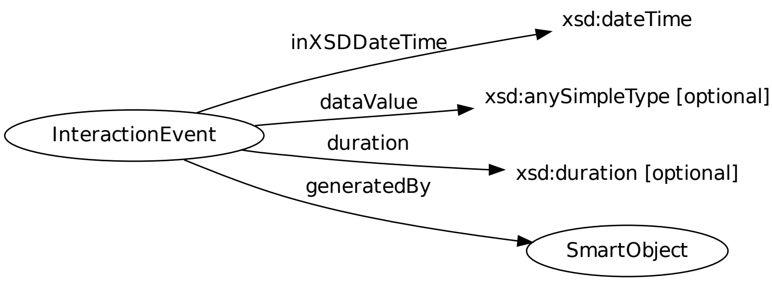


Figure 48: An interaction event as modelled in the ontology

An example of an event generated when an alarm is set is

```
:event-43495d51-29e3-11b2-807e-ac78eefc1f82
  rdf:type :AlarmSetEvent ;
  :generatedBy :phone1 ;
  :inXSDDateTime "2012-01-17T11:22:06.887+01:00"^^xsd:dateTime ;
  :dataValue "2012-01-17T12:00:00+01:00"^^xsd:dateTime .
```

DUL	EO	LODE	INTERACTION EVENTS
isObservableAt	time	atTime	inXSDDateTime
	place	inSpace	
hasLocation		atPlace	
hasParticipant	factor	involved	
involvesAgent	agent	involvedAgent	generatedBy
			dataValue

Table 4: Mappings between the various event models (adapted from [107])

A mapping between our interaction event model and the other event ontologies is shown in Table 4. Note that we do not model people or places in the current version of the ontology, as we consider these entities to be optional when describing interaction events.

The `duration` property is used to define the length of event. For example to increase the brightness of a lamp, we can generate an event to increase a value to a set maximum over a time period:

```
:event-43495d51-29e3-11b2-807e-ac78eefc1f83
  rdf:type :IncreaseLevelEvent ;
  :generatedBy ie:wakeup1 ;
  :inXSDDateTime "2012-01-17T11:23:06.887+01:00"^^xsd:dateTime ;
  :dataValue 255 ;
  :duration "PT3S"^^xsd:duration .
```

We consider interaction events to be identifiable and traceable. Each interaction event has an associated timestamp and a unique ID that is generated when the event occurs. All interaction events that occur are stored in the triple store.

Intentional, incidental and expected interactions were introduced in Section 2.5.2.

An intentional interaction, like pressing a light switch, is an interaction event if the light switch shares this information with other devices. Incidental or expected interactions, like the light turning on if the presence sensor is triggered, are also interaction events. System events, like a `TimeSetEvent`, which are invisible to the user are not considered to be interaction events.

8.2.1 *System events*

When a smart object first subscribes to the smart space, it specifically listens for events that are generated by other smart objects connected to it. This means that we also need some way of distributing system-wide events that all devices listen for. As an example, consider the `TimeSetEvent`. When the user sets the time on one device, we want the time to be immediately updated on all the other smart objects in the smart space, even if they are not connected to the device that generated the `TimeSetEvent`. If we define `TimeSetEvents` as a subset of `SystemEvents`, each smart object only need to subscribe to events of type `SystemEvent`.

8.2.2 *Feedback*

When setting an alarm for example, augmented feedback should be provided on all devices. Functional feedback, i.e. the alarm sound when an alarm fires is triggered, is delayed. This means that augment functional feedforward should be provided. We thus define two types of feedback events:

Preview events and indicator events were first introduced in Section 5.4.1.1.

- `PreviewEvent` - generated when a possible connection is being explored, displaying the possible functionalities enabled by the connection, i.e. augmented functional feedforward.
- `IndicatorEvent` - augmented feedback when smart object is connected and there is no immediate functional feedback, e.g. a sink "beeping" when the alarm is set on the source; used to confirm actions.

The type of feedback required depends on the functionality of the connection. It is important for the feedback to coincide in time and modality with the event generated, as to maintain the causal link that is perceived by the user.

The device used to make the connection, for example the Connector object, creates a temporary connection to the devices to be connected in order to generate a `PreviewEvent`. This `tempConnectedTo` property is a sub-property of the `connectedTo` property. This means that the smart objects will handle it as if it is a regular connection, and when the Connector object removes the `tempConnectedTo` relationship, the inferred `connectedTo` relationship will disappear as well.

8.2.3 *Discussion & Conclusion*

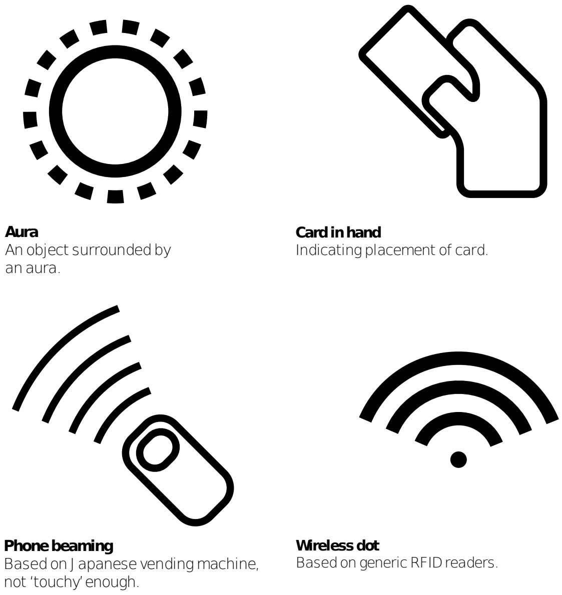


Figure 49: Examples from Arnall's "A graphic language for touch" (adapted from [4])

Tungare et al. [110] defined a *task disconnect* as "the break in continuity that occurs due to the extra actions outside the task at hand that are necessary when a user attempts to accomplish a task using more than one device." Kuniavsky [66] states that consistency is the key aspect in creating task continuity across devices, and that *interaction vocabularies* have recently emerged as a way of consistently interacting with a range of devices. This ranges from simple vocabularies of light patterns and motion as used by the (now discontinued) Nabaztag Internet-connected rabbit that could compose "sentences" with more complex meaning, to a set of visual icons by Timo Arnall [4] that represents various kinds of touch-based RFID interactions.

These interaction vocabularies try to smooth over task disconnects through consistency. We argue that by having a vocabulary, or ontology, of interaction events could improve consistency in ubiquitous computing environments. This is the intention of the work in this chapter — to provide an ontology of interaction events that improves consistency for users, as well interoperability between devices.

Arnall's touch-based vocabulary is shown in Figure 49. Arnall categorises his vocabulary of visual icons into four classes of interactions:

- Circles, of which the "Aura" is an example
- Card, of which the "Card in hand" is an example
- Wireless, of which the "Wireless dot" is an example
- Mobile, of which the "Phone beaming" is an example

The "Aura" icon communicates both the near-field communication capabilities of the technology, but also indicates that the physical object has capabilities beyond its form. The icons with cards or mobiles could improve consistency for a wide range of users which use these technologies on a daily basis.

Arnall's work is specific to touch-based RFID interactions, and helps to provide users with feedforward of device functionality that might not be immediately apparent otherwise. An ontology of interaction events, on the other hand, provides designers of smart objects with a vocabulary to describe the events that are shared between devices in order to improve interoperability and consistency.

9 ONTOLOGY ENGINEERING

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

— Antoine de Saint Exupéry [60], writer and poet

This chapter deals with the ontology- and reasoning-related aspects of the work in more detail. First, an overview of the different types of ontologies is provided, followed by a description of how semantic reasoning is performed with ontology languages. Our implementation, with reasoning performed using OWL and SPIN, is described. A set of 10 ontology design patterns that were identified during the three design iterations forms the second part of this chapter.

An ontology is a representation of knowledge (facts, things, etc.) in terms of concepts within a specific domain, as well as the relationships between them. Ontologies make it easier to publish and share knowledge. They are both machine-readable and human-understandable. The power of ontologies lies in their ability to create relationships among classes of objects, and to assign properties to those relationships that allows us to make inferences about them [61].

The word ontology is used in the literature to mean different things:

- a formal specification of concepts and relations in a domain, using axioms to specify the intended meaning
- an informal specification using UML class diagrams or entity-relationship models
- a vocabulary, or collection of named concepts agreed on by a group, defined in natural language

An example of clear conceptual modelling using roles is shown in Section 9.5.1.

What these different usages of the word have in common is that an ontology is a *community contract* about the representation of a domain [55]. It also has to be maintained during its lifespan, and is created through clear conceptual modelling based on philosophical notions.

An OWL file can be used to represent an ontology or the individuals (instances) it describes, or both the ontology and its

RESTRICTION	DL	MANCHESTER	OWL
Existential	$\exists$	`some`	`owl:someValuesFrom`
Universal	$\forall$	`only`	`owl:allValuesFrom`
Value	$\ni$	`value`	`owl:hasValue`
Equivalence	$\equiv$	`equivalentTo`	`owl:equivalentProperty,` `owl:equivalentClass`
Cardinality	$=$	`exactly`	`owl:cardinality`
Minimum cardinality	$\geqslant$	`max`	`owl:minCardinality`
Maximum cardinality	$\leqslant$	`min`	`owl:maxCardinality`

Table 5: OWL restriction definitions using different syntaxes: Description Logic, Manchester OWL Syntax[35] and OWL syntax

instances can be contained within the same file. For example, the concept `Man` could be defined as part of the ontology, and the individual `Gerrit` would be an instance of `Man`. The different types of restrictions that can be defined in OWL are shown in Table 5, together with the various syntaxes that can be used to represent these restrictions.

Even without using a reasoner to infer new facts, an ontology improves the usefulness of the data. Using unique identifiers to represent concepts and relationships enables a computer to find and aggregate new information. For example, the relationship `knows` in the FOAF ontology can be used to find and aggregate relationships between two individuals, where asserting

```
:Jun :knows :Gerrit .
:Gerrit :knows :Bram .
```

we can infer that

```
:Jun :knows :Bram .
```

We distinguish between four layers of ontologies, that are used to present concepts ranging from the more general to the more specific: foundational ontologies, core ontologies, domain ontologies and application ontologies.

9.1 LAYERS OF ONTOLOGIES

9.1.1 *Foundational ontologies*

Foundational or upper ontologies are aimed at modelling very basic and general concepts, as to be highly reusable in different scenarios [101]. They are used to align concepts in other ontologies, and to ensure consistency and uniqueness of these concepts. Examples of foundational ontologies include DOLCE, Basic Formal Ontology (BFO), OpenCyc and Suggested Upper Merged Ontology (SUMO). These ontologies can serve as reference ontologies when a new ontology is developed.

9.1.2 *Core ontologies*

Core ontologies are used to model knowledge about a specific field. A core ontology is based on a foundational ontology and should be modular and extensible [101]. A number of core ontologies exist for modelling things like events and multimedia objects. Core ontologies refine foundational ontologies by adding field-specific concepts and relations. The Event-Model-F ontology, for example, is used to model the causality, correlation and interpretation of events, and is based on DUL. Core ontologies achieve modularity and extensibility by following a pattern-oriented approach. Event-Model-F uses the DnS and Information Object patterns provided by DUL.

The Core Ontology Multimedia (COMM) ontology is used represent multimedia objects such as images, video and audio, and is also based on DUL. An audio recording could be modelled as `AudioData`, while a text description could be modelled as `TextData`. However, `AudioData` (a subconcept of DUL `InformationObject`) represents the information that is contained in the audio recording, not the digital audio stream itself [101]. The location of the audio file is represented with a concept that denotes the URI.

9.1.3 *Domain ontologies*

Domain ontologies represent reusable knowledge in a specific domain and are usually handcrafted. The Gene ontology, for example, describes gene products in terms of their biological processes, cellular components, and molecular functions in a species-independent manner [61].

Ontologies are particularly well-suited to domains such as biomedical research, where there is an abundance of available data with non-hierarchical relationships.

9.1.4 *Application ontologies*

An application ontology is created for a specific application, so they are not considered to be reusable. However, the tools or processes used to create the ontology may be reusable. The Cell Cycle ontology[1], for example, is specific to modelling the cell cycle process, a rather specialised domain.

9.2 OUR APPROACH

In the following sections we will describe our approach to modelling ontologies, as well as ontology design patterns that we have identified. First we introduce the features of OWL that we used. OWL can be used to define classes and relationships, as well as restrictions. A restriction is used to define a formal description of a class that restricts class membership [3].

An example of an OWL restriction is shown in Section 9.3.5.

In some cases we need more expressiveness than what is allowed by OWL. Rule languages go beyond what can be expressed by OWL, or can be easier to understand [54]. We made use of SWRL in the first design iteration, and in some of the examples in this chapter. Later, we discovered some limitations of SWRL, like not being able to construct new individuals. We also experienced some performance issues when using SWRL.

SWRL performance issues are described in more detail in Section 4.5.

This necessitated the switch to another way of defining rules, called SPIN. SPIN allows us to specify rules in SPARQL. These SPARQL rules are contained within the ontology itself. The TopSPIN reasoning engine, implemented in our version of the SIB, supports both OWL 2 and SPIN.

9.3 REASONING WITH OWL

In order to make the data generated by the smart environment more useful, we need a consistent way of understanding the combination of data from multiple sources. Reasoning or inferencing provides a robust solution to understanding the meaning of novel combinations of terms [54]. A reasoner may be used for truth maintenance, belief revision, information consistency and information creation in an information space [88].

As of October 2009 the OWL 2 Web Ontology Language is the W3C recommendation for creating ontologies. Most semantic

1 http://www.CellCycleOntology.org

reasoners have some kind of support for OWL as well as support for a rule language like SWRL:

- Pellet (Java): Supports OWL 2 and SWRL (DL-safe rules), has a command-line option with `explain` command.
- Fact++ (C++): Supports OWL DL, does not fully support OWL 2.
- HermiT (Java): Supports OWL 2 and SWRL (DL-safe rules without built-ins), uses hypertableau calculus to perform reasoning, comes pre-installed with Protégé editor, has a command-line option.
- TopSPIN (Java): Supports OWL 2 RL/RDF Rules defined as SPIN rules, comes pre-installed with TopBraid Composer.

Let us now look at a number of services provided by reasoners.

9.3.0.1 *Subsumption testing*

One of the services provided by a reasoner is to test whether or not one class is a subclass of another class, also known as subsumption testing. The descriptions of the classes are used to determine if a superclass/subclass relationship exists between them. It also infers disjointness and equivalence of classes. By performing such tests on the classes in an ontology it is possible for a reasoner to compute the inferred ontology class hierarchy. The reasoner can also determine class membership for individuals based on their properties, i.e. class membership does not always have to be asserted. It is also possible to infer new property relations with other individuals.

Subsumption refers to the reflexive, transitive and antisymmetric relationship between classes, that states that a class A subsumes a class B if and only if the set of instances of class A includes the set of instances of class B [93]. The same principle holds for OWL properties.

Preuveneers and Berbers [93] evaluated the Pellet ontology reasoner on a smart phone for semantic matching, but it was considered unsuitable due to performance requirements. They developed an encoding scheme to provide a compact representation of subsumption relationships. It is based on the idea that subsumption of classes in an ontology is somewhat related to multiple inheritance in an object-oriented programming language, which means that inheritance-encoding algorithms can be used for subtype testing. However, the algorithm cannot test for satisfiability - whether instances of a specific class can actually exist.

Being able to use a reasoner to automatically compute the class hierarchy is one of the major benefits of building an ontology using OWL. When constructing large ontologies the use of a reasoner to compute subclass-superclass relationships between classes becomes almost vital. Without a reasoner it is very difficult to keep large ontologies in a maintainable and logically correct state.

With ontologies it is possible for a class to have many superclasses, also called multiple inheritance. Usually it is easier to construct the class hierarchy as a simple tree, and leave computing and maintaining multiple inheritance to the reasoner. Classes in the asserted hierarchy therefore have no more than one superclass. This helps to keep the ontology in a maintainable and modular state and minimises human errors that are inherent in maintaining a multiple inheritance hierarchy. For example, in our ontology we have

```
AlarmSetEvent rdfs:subClassOf SetEvent .
TimeSetEvent rdfs:subClassOf SystemEvent .
```

where `TimeSetEvent` is asserted to belong to one superclass `TimeSetEvent`, but could also be inferred by the reasoner to belong to `SetEvent` if that is preferred.

9.3.1 *Consistency checking*

A reasoner performs consistency checking to check whether all axioms and assertions are consistent. Based on the description of a class the reasoner can check whether or not it is possible for the class to have any instances. A class is deemed to be inconsistent if it cannot possibly have any instances.

9.3.2 *Necessary versus necessary and sufficient*

RDF Schema (RDFS) is a subset of OWL [3].

A *necessary* condition will allow a class to be inferred as a subclass (`rdfs:subClassOf`), compared to a *necessary and sufficient* condition, which will make a class equivalent to another class (`owl:equivalentClass`). The second condition usually requires an intersection of classes to be defined using the `and` keyword.

9.3.3 *Inverse properties*

If one defines a new inverse property of an existing property with a specified domain and range, the inverse domain and range will be inferred for new individuals with this property. As an example:

Note that in Protégé this inverse domain and range might not show up for the property itself, but that it will be inferred for new individuals.

SmartObject ≡ isSmartObject ∃ `Self`

This can also be represented in OWL as:

```
:SmartObject
    a owl:Class ;
    owl:equivalentClass
    [   a owl:Restriction ;
        owl:hasSelf "true"^^xsd:boolean ;
        owl:onProperty :isSmartObject
    ] .
```

Any individual that is related to itself via the `isSmartObject` property will be identified as an instance of `SmartObject`, and any individual asserted as an instance of SmartObject will be related to itself via that property [57].

9.3.4 *Property chains*

A new feature introduced in OWL 2 is property chains, which allows for the specification of the propagation of a property along some path of interconnected properties [58]. Examples of property chains are shown in Section 4.4.2 and Section 7.4.

9.3.5 *Using cardinality restrictions*

When modelling cardinality in OWL 2, one might expect to be able to infer that an individual is a member of a class based on a cardinality restriction, for example

```
Class: TwoButtonDevice

        SubClassOf:
                Device hasButton exactly 2 Button
```

Unfortunately, due to the OWA, it cannot be known whether an individual might have additional properties of that type. The only way to identify an individual is using minimum cardinality. However, this approach can be problematic if the concept is underspecified [58].

In OWL 2, it is possible to define a Qualified Cardinality Restriction (QCR), which means the cardinality restriction can be applied to a specific class [54].

This means that it is possible to define that a smart object has only one current state:

```
SmartObject
rdfs:subClassOf
            [
              rdf:type owl:Restriction;
              owl:qualifiedCardinality 1;
              owl:onProperty hasCurrentState;
              owl:onClass State
            ];
```

If we then assert a certain smart object to have two current states, e.g.

```
phone1 hasCurrentState playing .
phone1 hasCurrentState stopped .
```

Individuals are distinct if it is asserted that they are different from one another.

it will violate the QCR if `playing` and `stopped` are distinct. In earlier versions of OWL, it was not possible to define a specific class for a cardinality restriction.

9.4 REASONING WITH SPIN

SPIN[2] is a W3C Member Submission created and maintained by TopQuadrant, who is also responsible for the TopBraid Composer ontology editor. With SPIN, rules are expressed in SPARQL, the W3C recommended RDF query language, which allows for the creation of new individuals using CONSTRUCT queries. Let us now look at some features of SPIN.

9.4.1 *Integrity constraints*

SPIN allows us to specify integrity constraints, e.g. that

```
:event1 :generatedBy :device1 .
```

2 http://www.spinrdf.org

should exist. Domain and range are not integrity constraints, but allow us to infer for example the class type of new individuals, e.g. if

```
:generatedBy rdfs:range :SmartObject .
```

then asserting

```
:event1 :generatedBy :device1 .
```

would infer

```
:device1 rdf:type :SmartObject .
```

9.4.2 *SPARQL Rules*

SPIN allows for fine-grained control of how rules are executed. For example, it is possible to have a rule fire only once, by setting the SPIN property `spin:rulePropertyMaxIterationCount` to 1, in cases where new inferences could cause the rule engine to iterate infinitely. It is also possible to specify the order in which rules are executed using `spin:nextRuleProperty`.

9.4.3 *Built-in SPARQL Functions*

SPIN has a number of built-in functions[3] that provides additional functionality not available in OWL 2. These built-in functions can be very helpful when creating your own SPIN rules, functions or magic properties. They can be used to retrieve substrings (`fn:substring`), perform modulo arithmetic (`spif:mod`), or generate random numbers (`spif:random`).

Built-in functions with `fn:` (XPath/Xquery) or `afn:` (ARQ Functions) prefix are also available as part of ARQ, the Jena query engine. The `spif:` prefix denotes the SPIN Standard Functions Library.

An example of where they are used in our ontology is the `afn:now()` function in the `currentDateTime` magic property:

```
SELECT ?datetime
WHERE{BIND(afn:now() AS ?datetime) .
}
```

Magic properties are described in Section 9.4.5.

Some built-in functions, like `spif:buildUniqueIRI` (used to create new URIs), are only available as part of the extended TopBraid SPIN API[4], and cannot be used with the free open-source

3 The reference documentation for the built-in functions can be accessed in TopBraid Composer from `Help` → `Help Contents` → `TopBraid Composer` → `Reference` → `SPARQL Functions Reference`

4 Available under a commercial license from TopQuadrant

edition[5]. That said, it is possible to build your own `buildURI` function using `fn:concat` as we did in the second design iteration:

```
BIND (IRI(fn:concat("example.com#mediaPath_", afn:localname(?this),
"_to_", afn:localname(?x3))) AS ?mp) .
```

9.4.4 *Custom functions*

It is possible to create your own custom functions in SPIN. These functions are written in SPARQL and stored in the ontology. An example of a custom function we built[6] is `getMaxDateRsc`, which is used to retrieve the last interaction event that was generated by a specific smart object:

If you use the `.spin.rdf` extension to store the ontology file, custom functions will be loaded into TopBraid Composer on startup.

```
SELECT ?lastEvent
WHERE{
    ?lastEvent events:generatedBy ?arg1 .
    ?lastEvent events:inXSDDateTime ?last .
}
ORDER BY DESC (?last)
LIMIT 1
```

This was then combined with a SPIN rule to create an object for the `hasLastEvent` property:

```
CONSTRUCT{
    ?this events:hasLastEvent ?lastEvent .
}
WHERE{BIND (events:getMaxDateRsc(?this) AS ?lastEvent) .
}
```

The SPIN rule is required as magic properties cannot be used in local restrictions on their own.

When loading an ontology with SPIN functions into Jena, the functions should be registered using

```
SPINModuleRegistry.get().registerAll()
```

An extension of SPIN, called SPINx, allows for the definition of more elaborate custom functions using JavaScript. Unfortunately it cannot access the triple graph at execution time, but it does operate on arguments. Jena allows similar functionality to SPIN and SPINx functions using a `FunctionFactory`, which allows you to define and register your own functions in Java.

5 http://topbraid.org/spin/api/
6 With help from Scott Henninger and Holger Knublauch from TopQuadrant

9.4.5 *Magic properties*

Magic properties, also called property functions, may be used in SPIN to dynamically compute values, even if there are no corresponding triples in the model. For example, we created the magic property `currentDateTime` with the SPIN body

The inferencing engine does not always infer superclasses for SPARQL queries, which could cause problems for magic properties.[8]

```
SELECT ?x
WHERE{BIND (afn:now() AS ?x) .
}
```

When we now create a query for something like

```
:phone1 :currentDateTime ?date
```

the current date/time is returned as an object. This allows us to write KP queries at triple-level, without having to send a SPARQL query from the KP to the SIB. Magic properties are more flexible than SPIN functions and can return multiple values.

9.5 ONTOLOGY DESIGN PATTERNS

In software engineering, design patterns are generalised solutions to problems that commonly occur in a specific software context. An example of such a pattern is the observer pattern, in which a software object maintains a list of observers which are notified of state changes. The observer pattern is one of the original patterns described in the seminal book on design patterns by Gamma et al. [45]. The blackboard pattern, used in our software architecture, is a generalised version of the observer pattern that allows multiple readers and writers.

The blackboard pattern was first mentioned in Section 1.2.5.

A similar approach to design patterns has been applied to ontologies [47, 56, 33]. Dodds and Davis [33] used the following pattern template to document an ontology design pattern in their book "Linked Data Patterns":

- Question - A question indicating the problem the pattern is designed to solve
- Context - Description of the goal and context of the pattern
- Solution - Description of the pattern
- Example(s) - Real-world implementations that make use of this pattern

- Discussion - Analysis of the pattern and where it can be used
- Related - List of comparable patterns

They formalised a number of linked data patterns into a pattern catalogue, and we will now use the same pattern template to describe ontology design patterns that can be applied in the context of smart environments. In this section we first look at three examples of existing ontology design patterns, before we focus on new patterns that were identified during the course of the work described in this thesis.

One of the example patterns, DnS, is an ontology design pattern provided by the Ontology Design Patterns (ODP) initiative[9]. They maintain an entire online library of ontology design patterns, to be used as building blocks for creating new ontologies.

ODP distinguishes between a number of different pattern types, including:

- Content patterns, e.g. the Role pattern that defines `Student` as a role instead of a subclass of `Human`
- Logical patterns, like the n-ary relation or Situation pattern
- Reengineering patterns, e.g. converting microformats to RDF
- Alignment patterns, e.g. aligning FOAF with the VCard format
- Anti-patterns, e.g. modelling `City` as a subclass of `Country`

The first example pattern below, called the Role pattern, is required reading for understanding the DnS pattern.

9.5.1 *The Role pattern*

How can we represent the roles of devices and agents in an ontology?

9.5.1.1 *Context*

An example of clear conceptual modelling is that a `Student` is not a subclass of `Human`, but a *role*.

9 http://ontologydesignpatterns.org/wiki/Submissions:DescriptionAndSituation

9.5.1.2 *Solution*

Roles can be modelled as classes, individuals or properties.

9.5.1.3 *Example(s)*

Roles can be modelled as classes:

```
Object rdf:type Role .
```

or as individuals:

```
Jim rdf:type Person .
SongWriter rdf:type Role .
Jim hasRole SongWriter .
```

or even as properties:

```
Table legs Books .
```

where books are being used in the role of table legs.

9.5.1.4 *Discussion*

A commonly occurring issue when modelling ontologies is to whether model the concept as a property or a class. Consider the role *student*, where `Mark` can be seen as either an individual of the `Student` class, or have a relationship via a `student` property with his university. Classes have stronger ontological commitment[10] than properties, but using properties are often more convenient for practical use [57]. OWL 2 punning allows an entity to be treated as both a property and a class without comprising ontological commitment.

9.5.1.5 *Related*

- The Role pattern is described in detail in Hoekstra's PhD thesis [56]
- The Time Indexed Person Role Pattern [47]

9.5.2 *Descriptions and Situations (DnS) pattern*

How do we model non-physical objects like plans, schedules and context in an ontology?

10 See Section 11.2.1

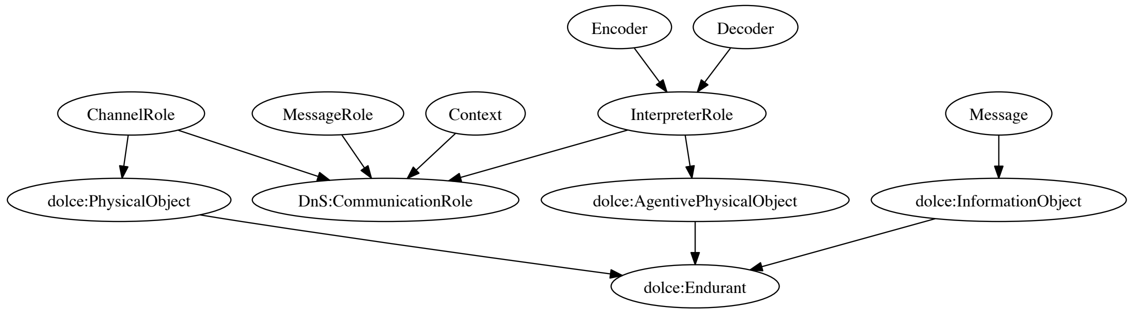


Figure 50: Example of modelling communication theory using DnS and DOLCE

9.5.2.1 *Context*

While modelling physical objects using an ontology is relatively straightforward, it becomes non-trivial when modelling *non-physical objects* [46] such as plans, schedules, social constructs, etc. Existing theoretical frameworks like BDI theory [18] are not at the level of concepts or relations, which we need to be able to model non-physical objects as a set of statements. The DnS pattern grew out of the work done on the DOLCE ontology to solve this problem.

During a summer school attended by the author, Aldo Gangemi (co-creator of DOLCE) mentioned that he considers DOLCE to be a collection of ontology design patterns.

9.5.2.2 *Solution*

The DnS design pattern provides an ontological formalisation of context [101]. It achieves this by using *roles* to classify entities into a specific context. The pattern defines a *situation* that satisfies a *description*. The `describes` object property is used between a `Description` and an object, while the `satisfies` object property relates a `Situation` with a `Description`.

9.5.2.3 *Example(s)*

As an example, consider communication theory [106] as modelled with DnS in Figure 50, where there is an encoder, a message, a context[11], a code and channel. In DnS, the encoder and decoder are modelled as agentive physical objects in DOLCE, while the channel is a non-agentive physical object. Messages are considered information objects.

11 What the message is about, not the circumstances surrounding the communication

9.5.2.4 *Discussion*

With DnS one can also reify events and objects and describe the n-ary relation that exists between multiple events and objects.

9.5.2.5 *Related*

- The DUL ontology [47]

9.5.3 *Defining n-ary relations*

How do we represent relations among more than two individuals?

9.5.3.1 *Context*

In OWL, a property is a binary relation between two individuals. However, some relationships are not binary and involve more than two resources, for example when modelling events.

9.5.3.2 *Solution*

We can use n-ary relations [86] to model relationships between more than two resources. A class is created to represent the relationship, with an instances of the class used to represent the relationship between the various resources.

9.5.3.3 *Example(s)*

`event-43495d51-29e3-11b2-807e-ac78eefc1f83` is an example of an `Event` instance that represents the n-ary relation between the device `phone1` and the various event resources:

```
:phone1 :generatesEvent :event-43495d51-29e3-11b2-807e-ac78eefc1f83.

:event-43495d51-29e3-11b2-807e-ac78eefc1f83
  rdf:type :IncreaseLevelEvent ;
  :inXSDDateTime "2012-01-17T11:23:06.887+01:00"^^xsd:dateTime ;
  :dataValue 255 ;
  :duration "PT3S"^^xsd:duration .
```

9.5.3.4 *Discussion*

This pattern is commonly used to represent complex relationships. This is quite a powerful pattern, as it can also be used to define the temporal order of sequences [86].

9.5.3.5 *Related*

- Qualified Relation pattern [33]

9.5.4 *Naming interaction events*

How should the URI of an interaction event be structured so that the name forms a natural hierarchy?

9.5.4.1 *Context*

Interaction events tend to form natural groups, such as events related to a specific device class. Reflecting these groups in the name of the interaction event itself makes it easier for developers to understand existing and/or inferred groupings, and to classify new events into an existing hierarchical event structure.

9.5.4.2 *Solution*

We use the notation

```
[DeviceClass][Action]Event
```

to define the interaction event.

9.5.4.3 *Example(s)*

Consider a simple light switch with two states, `Up` and `Down`. We can define two interaction events, `switchDownEvent` and `switchUpEvent`, which can then later be grouped by either device class or by action.

9.5.4.4 *Discussion*

If the naming convention of a URI follows a common pattern, they become easier to remember and easier to work with. They can even be constructed automatically. It makes the URI human-readable and improves the relation between the name and the event it describes.

9.5.4.5 *Related*

- Hierarchical URIs [33]
- Patterned URIs [33]

9.5.5 *Using local reflexivity in property chains*

How can we specify classes as part of an OWL 2 property chain?

9.5.5.1 *Context*

Sometimes it is necessary to restrict property chains to specific classes. We need to be able to specify these classes as part of the property chain.

9.5.5.2 *Solution*

The `self` keyword[12] is used to indicate local reflexivity (also called a self restriction) in OWL 2 and can be used to transform classes to properties when creating property chains.

9.5.5.3 *Example(s)*

We can apply local reflexivity to the class `Student`, for example

$$\text{Student} \equiv \text{isStudent}\ \texttt{some self}$$

If the individual Mark has a `isStudent` relation with itself, it will be inferred that Mark is a `Student`. Also, if Mark is asserted as a `Student`, then the `isStudent` property will be inferred. This can then be combined with property chains where necessary, e.g.

$$\text{hasRole} \circ \text{isStudent} \sqsubseteq \text{student}$$

9.5.5.4 *Discussion*

In his PhD thesis on ontology design patterns, Hoekstra [56] uses this pattern extensively to model actions, beliefs, intentions and social constructs. For example,

$$\begin{aligned} \text{Intention} &\equiv \text{isIntention}\ \texttt{some self} \\ &\sqsubseteq \text{PropositionalAttitude} \\ \text{holds} \circ \text{isIntention} \circ \text{towards} &\sqsubseteq \text{intends} \end{aligned}$$

12 Manchester syntax, used when editing ontologies in Protégé and other ontology editors. See Table 5.

9.5.5.5 *Related*

- DnS pattern

9.5.6 *Semantic matching with property chains*

How can we perform semantic matching of functionalities between devices using property chains?

9.5.6.1 *Context*

Property chains are useful for semantic matching, but with basic property chains the inverse is inferred as well, which is not always desired. Property chains cannot be made irreflexive, as only *simple* properties can be irreflexive in order to guarantee decidability [8]. Defining domain and range to as constraints just makes the ontology inconsistent. Thus, when using property chains, the properties involved need to be symmetric, as in

hasFunctionality $\circ$ isFunctionalityOf

9.5.6.2 *Solution*

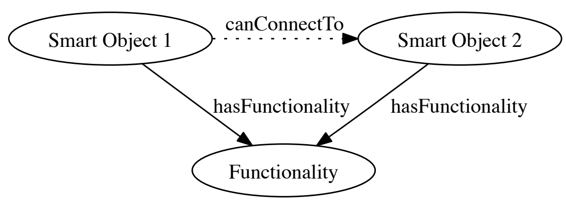


Figure 51: Two individuals related to the same object

When we have two individuals with the same object, but different predicates (see Figure 51), and we want to infer a new property, this is intuitively represented in SWRL:

```
hasFunctionality(?s1,?f), hasFunctionality(?s2, ?f) ⇒ canConnectTo(?s1,?s2)
```

However, this cannot be represented in the same fashion using a property chain, as

hasFunctionality $\circ$ hasFunctionality $\sqsubseteq$ canConnectTo is not equivalent. This is however, easily solved by introducing an inverse property `isFunctionalityOf`, and the property chain becomes

$$\text{hasFunctionality} \circ \text{isFunctionalityOf} \sqsubseteq \text{canConnectTo}$$

Modelling the above using the Relation Partition Algebra (RPA) of Feijs [40], where
$\text{hasFunctionality} \circ \text{hasFunctionality}^{-1} \subseteq \text{canConnectTo}$

shows the property chain can also be represented using RPA, apart from the inverse relation, which is denoted by $R^{-1} = \{(x, y) | (y, x) \in R\}$.

9.5.6.3 *Example(s)*

First we define two smart objects and their corresponding functionalities:

```
:Music a :Functionality .

:phone1 a :SmartObject .
:phone1 :functionalitySource :Music .

:speaker1 a :SmartObject .
:speaker1 :functionalitySink :Music .
```

Using the property chain

$$\text{functionalitySource} \circ \text{isFunctionalityofSink} \sqsubseteq \text{canConnectTo}$$

where `isFunctionalityofSink` is the inverse property of `functionalitySink`, we can infer that

```
:phone1 :canConnectTo :speaker1 .
```

9.5.6.4 *Discussion*

There are two caveats when using property chains to perform semantic matching. First, OWL 2 property chains cannot be built with datatype properties, only object properties, i.e. use

```
:device1 :hasFunctionality :Audio .
```

instead of

```
:device1 :hasFunctionality "audio" .
```

This means we cannot infer

```
:device1 :hasRFIDTag "ABCD123F" .
```

SWRL was used for semantic matching in the second design iteration in Section 4.4.2. SPIN was used in the third design iteration, with the implementation described in more detail in Chapter 7.

and we have to use a rule language like SWRL or SPIN.

The second caveat is that property chains cannot be used for cardinality restrictions. We have only tested this with the Pellet reasoner, and it is possible that other reasoners could allow for this to happen.

9.5.6.5 *Related*

- The Role pattern

9.5.7 *Inferring new individuals*

How can new individuals be created when an existing literal value changes?

9.5.7.1 *Context*

SWRL built-in atoms in rule heads [54] present another solution to this problem, but these built-in atoms cannot be handled by reasoners like Pellet, which only supports DL-safe rules.

Ontology languages like OWL are used to classify existing individuals, not create new ones. In some cases we want to insert a new individual when a literal value changes or is inserted. When using only OWL and DL-safe rules (e.g. SWRL), no new individuals may be inserted, and the work-around is that individuals are pre-populated in the triple store. For example, if `OnEvent` and `OffEvent` are pre-populated, you can model that

```
:event1 :dataValue 1 .
```

should infer

```
:event1 :mappedTo :OnEvent .
```

9.5.7.2 *Solution*

A SPARQL CONSTRUCT query, defined as a SPIN rule, can be used to insert a new individual into the triple store.

9.5.7.3 *Example(s)*

A new individual, representing a media path, can be inferred using:

```
CONSTRUCT{
    ?mp a sc:MediaPath .
    ?x3 sc:hasMediaPath ?mp .
    ?mp bonding:mediaSourceSO ?x2 .
    ?mp bonding:mediaOriginator ?this .
}
WHERE{
    ?this sc:convertsMediaType ?x2 .
    ?x2 sc:convertsMediaType ?x3 .
    ?this sc:connectedTo ?x3 .
    BIND (IRI(fn:concat("example.com/ontology#mediaPath_",
     afn:localname(?this), "_to_",afn:localname(?x3))) AS ?mp) .
}
```

In the example, a new `mediaPath` individual is created if two smart objects are connected to each other and there is a `mediaSourceSO` (semantic transformer) that converts the media types between them. This could be a media player transmitting music as source, an ambient lighting object that accepts RGB colour values as sink, and a semantic transformer that converts audio streams into RGB lighting information. For more information about media paths and semantic transformers, see [81].

The `?this` variable indicates to SPIN how the definition should be applied to the members of a class, as the rule itself is defined as part of the class definition - thus defining the scope of the query. `fn:concat` and `afn:localname` are SPIN functions used to concatenate the name of the individual and retrieve the local names of the variables used respectively.

9.5.7.4 *Discussion*

When a new individual is inserted using a SPIN rule, care should be taken in how the name of the individual is generated. If we define the new individual as a blank node, the TopSPIN reasoning engine will not terminate, because a new blank node is defined with each iteration. The same issue arises if we assign a random value as the name. Using a fixed URI is a simpler solution, as shown in the example above.

9.5.7.5 *Related*

None.

9.5.8 *Removing inferred triples*

How do we remove inferred triples from the triple store when an asserted triple changes?

9.5.8.1 *Context*

Removing inferred triples when an asserted triple changes, or is deleted from the model, can be notoriously difficult. For irreflexive properties, it is possible to use constraint violations to detect them, and then remove them one by one. Unfortunately constraint violation checking is very slow, for example taking 834 ms when the inferencing itself takes only 313 ms[13]. Creating a SPIN rule to clean up irreflexive properties does not work, as the properties get inserted and removed after each iteration of the inference engine.

9.5.8.2 *Solution*

Two models are used in the triple store, one for the asserted model and one for the inferred model. The inferred model is cleared before each reasoning iteration.

9.5.8.3 *Example(s)*

Not applicable.

9.5.8.4 *Discussion*

According to TopQuadrant[14], removing inferred triples based on a triple that was deleted is a tricky use case, requiring a `BufferingGraph` that is not available in the open source SPIN API.

9.5.8.5 *Related*

None.

9.5.9 *Inferring subclass relationships using properties*

Can we infer subclass relationships based on existing properties using OWL?

---

13 Based on a model size of 2304 inferred triples

14 `topbraid-users` mailing list discussion

9.5.9.1 *Context*

Suppose we wanted to use an object property called `mappedTo` to create a mapping between interaction events, for example

```
SwitchUpEvent mappedTo SwitchOnEvent .
```

This prompts the question: Is it possible to create an OWL restriction that says

If Class A is related via Property B to Class C, then Class A is a subclass of Class C.

When modelled in SPARQL, it looks like this:

```
CONSTRUCT{
        ?A rdfs:subClassOf ?C .
}
WHERE{
        ?A :B ?C .
}
```

9.5.9.2 *Solution*

Evidently, this could be implemented as a SPIN rule, but we would prefer an OWL-only solution. It turns out that while it is not possible in OWL 2 DL, it is possible in OWL 2 RL/RDF Rules:

```
:B rdfs:subPropertyOf rdfs:subClassOf .
```

9.5.9.3 *Example(s)*

To solve our original problem in the Context, we would define

```
mappedTo rdfs:subPropertyOf rdfs:subClassOf .
SwitchUpEvent mappedTo SwitchOnEvent .
```

which would then infer

```
SwitchUpEvent rdfs:subClassOf SwitchOnEvent .
```

9.5.9.4 *Discussion*

This simple but powerful pattern is a good example of meta-modelling.

9.5.9.5 *Related*

None.

9.5.10 *Inferring connections between smart objects and semantic transformers*

When we use semantic transformers to control devices, how can we infer these connections between the smart objects and the semantic transformer?

9.5.10.1 *Context*

In the sleep use-case, a semantic transformer was implemented in order to generate lighting values for the dimmable lamp to create the desired wakeup experience. During the implementation, several observations and decisions were made:

- Between smart objects and semantic transformers only `indirectlyConnectedTo` connections can exist, as the semantic transformers are virtual entities that cannot be directly connected to smart objects using the Connector object.
- When a `canIndirectlyConnectTo` relationship is inferred between smart object A and the semantic transformer B, and between B and smart object C, a `canConnectTo` relation between A and C should be inferred (transitive).
- When a connection is made between two smart objects that can be connected through a semantic transformer, the semantic transformer is connected to the smart objects with `indirectlyConnectedTo` relationships, and a `connectedTo` relationship between the smart objects is then automatically inferred.
- A semantic transformer thus acts as a bridge.
- A semantic transformer is *not* a smart object.

When using semantic transformers to control other smart objects, we could make use of the n-ary ontology design pattern, which was also applied to creating media paths in Section 4.4.2 on semantic matching:

- Subscribe to `controlSource` to see if it becomes a control source

- When it becomes a control source, subscribe to the events generated by the control originator

While this is feasible, it is complicated and we would like to use a simpler solution using `connectedTo` relationships. What we would like to infer is shown in Figure 52.

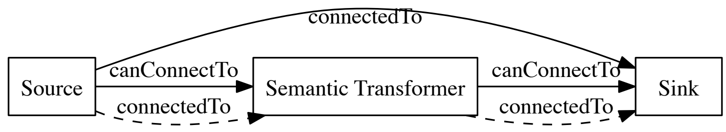


Figure 52: Inferring `connectedTo` relationships between sources/sinks and a semantic transformer

9.5.10.2 *Solution*

At first glance, it seems like this might be expressed using property chains and local reflexivity, as described in the ontology design pattern in Section 9.5.5.2. However, this is a special case which cannot be expressed in OWL. It can, however, easily be expressed as a SPIN rule as follows:

```
CONSTRUCT{
    ?source :connectedTo ?semanticTransformer .
    ?semanticTransformer :connectedTo ?sink .
}
WHERE{
    ?source :canConnectTo ?semanticTransformer .
    ?semanticTransformer :canConnectTo ?sink .
    ?source :connectedTo ?sink .
}
```

9.5.10.3 *Example(s)*

If the following triples are asserted:

```
:phone1 a :SmartObject .
:phone1 :functionalitySource :Alarm .

:lamp1 a :SmartObject .
:lamp1 :functionalitySink :AdjustLevel .
```

```
:wakeup1 a :SemanticTransformer .
:wakeup1 :functionalitySource :Alarm .
:wakeup1 :functionalitySink :AdjustLevel .

:phone1 :connectedTo :lamp1 .
```

Using the pattern defined in Section 9.5.6, we infer:

```
:phone1 :canConnectTo :wakeup1 .
:wakeup1 :canConnectTo :lamp1 .
```

Using this pattern, we infer the following `connectedTo` relationships:

```
:phone1 :connectedTo :wakeup1 .
:wakeup1 :connectedTo :lamp1 .
```

9.5.10.4 *Discussion*

RPA was first mentioned in Section 9.5.6.2.

In some cases, SPIN rules can be easier to compose and interpret than ontology restrictions and property chains. In this case, it cannot even be expressed in OWL, as OWL has no support for modelling property intersections. In RPA, on the other hand, relations are first class citizens, and Figure 52 can be composed using:

$$\text{connectedTo} \subseteq \text{canConnectTo} \cap \text{connectedTo} \circ \text{canConnectTo}^{-1}$$

9.5.10.5 *Related*

- N-ary pattern
- Semantic matching with property chains

9.6 DISCUSSION

When applying inference to the physical world, the level of ambiguity and uncertainty is quite high. A system might infer that you are in a room because your RFID badge is in a room. What if you forgot your badge in the office? The challenge is to figure out what functions in the smart home are possible with limited inference, which are possible only through inference, and which require an oracle [37]. Systems that rely on inference will be wrong some of the time, and users will need to have models to figure out how the system arrives at its conclusions, along with ways to override the system's behaviour.

Sabou [100] argues that smart objects will require more sophisticated reasoning mechanisms than what is currently used in the area of sensor networks, which primarily relies on subsumption matching. They expect that smart spaces will rely on rule engines rather than DL reasoners, and that the ambiguities and uncertainties in smart environments will require fuzzy or probabilistic methods.

Throughout the development of the ontology, we tried to avoid rule-based formalisms where possible, to see to what extent we can push the limits of OWL 2's expressive power. Hoekstra and Beuker [58] noted that to avoid problematic interactions between the two formalisms, it is undesirable to combine them. However, they also accepted that it is sometimes unavoidable, given the real problems that occur with elaborate concepts.

When performing semantic reasoning with a triple store, there is the potential for a combinatorial or state explosion problem. There are various methods that we used to avoid this problem. In cases where rules could cause the reasoning engine to iterate indefinitely, for example where new individuals were constructed, we ensured that the new triple representing the individual is not generated with a different URI after every iteration. Another strategy we used was to limit the complexity of the ontology itself.

It is our experience that people commonly underestimate the differences between data modelling and ontology engineering. While some concepts in an ontology can be modelled using UML class diagrams or represented using Java objects, there are some fundamental differences. Data modelling does not allow for axiomatisation to specify the semantics of the information, nor is it much concerned with conceptual modelling based on philosophical notions.

However, much is already gained with using some simple ontology engineering techniques, such as unique identifiers or distinguishing between actors and roles. As James Hendler, one of the authors of the seminal article on the Semantic Web in Scientific American [12], once stated, "a little semantics goes a long way".

10 SOFTWARE ARCHITECTURE

Interaction is an iterative process of listening, thinking, and speaking between two or more actors.

— Chris Crawford [83], game designer

Parts of this chapter have previously appeared in [80] and [82]

Existing architectural patterns for software like the MVC model, Document-View and Presentation-Abstract-Control are considered to be inadequate when trying to design software architectures in the ubiquitous computing domain. Ubiquitous computing needs new kinds of mechanisms to meet the flexibility needed to change the purpose, functionality, quality and context of a software system [79].

MVC was first mentioned in Section 2.3.

In this chapter the software architecture used in the three design iterations is described in more detail. It is quite a short chapter, as most of the software architecture issues have already been discussed in the three design iteration implementations in Sections 3.4, 4.4 and 5.4. However, we consider it important that the final software architecture design has its own dedicated chapter, so that it can act as a reference design for future implementations.

We first look at some characteristics of ubicomp middleware, followed by a discussion of the publish/subscribe paradigm and the blackboard architectural pattern. We then look at the Message-Oriented Middleware (MOM) implementation used within the SOFIA project, called SSAP. The rest of the chapter is dedicated to the two main implementations of the software architecture as used within the SOFIA project – Smart-M3 and ADK-SIB. These implementations are interoperable with one another through the use of SSAP.

10.1 CHARACTERISTICS OF UBICOMP MIDDLEWARE

There are a number of characteristics, or quality attributes, that are specific to middleware for ubiquitous computing, as defined by Niemelä and Vaskivuo [79]:

- Interoperability
- Scalability

- Reusability
- Maintainability
- Extensibility
- Portability
- Adaptability
- Survivability
- Agility
- Fidelity

Interoperability is defined as the ability for software applications written in different programming languages, running on different platforms with different operating systems, to communicate and interaction with one another over different networks. Scalability is the ability of the system to handle larger numbers of smart objects. Reusability, maintainability and extensibility are characteristics that consider the evolution of software systems. Portability and adaptability are important characteristics for software that has to work in a heterogenous system of devices and networks. Survivability is the ability of a system to timely deliver essential services in the face of attack, failure or accident. Agility is the sensitivity to changes in resource availability. Fidelity is defined to mean to degree to which data presented on a client matches the reference copy at the server.

We focused on a subset of these attributes while working on the software architecture, including interoperability, reusability, maintainability and extensibility. Interoperability was achieved by adhering to the SSAP specification, as described in more detail in Section 10.3. Using ontologies and other Semantic web technologies helped us to improve reusability, while elements of maintainability were tested using the Cognitive Dimensions framework, described in more detail in Chapter 11. Extensibility was achieved by modelling devices and their capabilities in such a way that other devices could easily be added to the system.

Potential scalability was tested by evaluating the performance of the software architecture, as described in Section 11.1.

10.2 PUBLISH/SUBSCRIBE PARADIGM AND THE BLACKBOARD PATTERN

In publish/subscribe systems, subscribers register their interest in a specific event, and are notified when this event occurs after a publisher publishes the event. The strength of the publish/-subscribe paradigm is that entities are decoupled in time, space and synchronisation [39]. Space decoupling means that the interacting entities do not need to be aware of each other. Time decoupling means that the entities do not need to participate in the interaction at the same time. Synchronisation decoupling means that subscribers can asynchronously be notified when an event occurs. Removing synchronisation dependencies between entities increases scalability.

There are three variants of publish/subscribe systems:

- Topic-based – Entities subscribe to individual topics, usually with some form of hierarchical addressing to organise the topics
- Content-based – Consumers subscribe to selective events by specifying filters, using some kind of subscription language
- Type-based – Events are filtered according to their type

In the SOFIA project, KPs communicate with a message broker using the blackboard architectural pattern, where the message broker uses a triple store as a common knowledge base. Communication between KPs occurs through the insertion and removal of triples into or from the triple store. Given a set of smart devices, the blackboard may be used to share information between these devices, rather than have the devices explicitly send messages to one another. If this information is also stored according to some ontological representation, it becomes possible to share information between devices that do not share the same representation model, and focus on the semantics of that information [88]. The SIB is the information store of the smart space, and contains the blackboard, ontologies, reasoner and required service interfaces for the KPs or agents.

This blackboard approach is complemented by a publish/-subscribe component, that allows KPs to subscribe to specific triples in the triple store. The KPs are then notified when these triples are added, removed or updated in the triple store. Communication between the KPs and SIB occurs using SSAP, which is the focus of the next section.

10.3 SMART SPACE ACCESS PROTOCOL (SSAP)

MOM is used to send messages between components in a distributed system. Commercial options include Java Message Service (JMS), Microsoft Message Queuing (MSMQ) and IBM's WebSphere framework. Advanced Message Queuing Protocol (AMQP) is an emerging standard, of which RabbitMQ[1] is a popular implementation. ZeroMQ[2], also written as ØMQ , was created to be simpler and faster than the AMQP standard, and does not require a dedicated message broker. Other message protocols include Extensible Messaging and Presence Protocol (XMPP), Message Queue Telemetry Transport (MQTT) and Streaming Text Oriented Messaging Protocol (STOMP).

XMPP is an Internet Engineering Task Force (IETF) standard.

In the SOFIA software architecture, KPs communicate with the SIB through SSAP messages [59] over TCP/IP. SSAP consists of a number of operations to insert, update and subscribe to information in the SIB. These operations are encoded using XML.

For operations initiated by a KP, the KP sends a request message and the SIB responds with a corresponding confirmation message. For SIB initiated operations, the SIB sends an indication message. The KP does not respond to SIB initiated operations, as indication messages contain non-essential information. Every session must start with a join operation, and a leave operation ends a session.

To insert information into the triple store, an insert operation is used by the KP, where the triples are encoded in RDF/XML. A SIB confirmation message indicate whether the operation was successful or not. Similarly, a remove operation is used to remove information from the triple store. An update operation removes information from the triple store and inserts new information as an atomic operation.

To query the triple store, a template consisting of a list of triples is used, where each triple may have a wildcard as its subject, predicate or object. The result of the query is a list of all triples that match the template. All triples in the triple store that match any of the triples in the list are returned.

A subscribe operation creates a persistent query that is stored in the SIB and is re-evaluated automatically after each change to the contents of the triple store. An unsubscribe operation will terminate a persistent query. The publish/subscribe mech-

1 http://www.rabbitmq.com/
2 http://www.zeromq.org/

anism used is closest in scope to the content-based variant described in the previous section.

SSAP is supported by both the SIB implementations used in our work, such that software developed for the one implementation is also interoperable with the other implementation. We now focus in more detail on these two implementations: Smart-M3 and ADK-SIB.

The Smart-M3 implementation was used during the first design iteration, while the ADK-SIB implementation was used during the second and third design iteration.

10.4 SMART-M3 ARCHITECTURE

The M3 (multi-device, multi-vendor, multi-domain) architecture is an interoperability platform based on a blackboard architectural model that implements the ideas of space-based computing [59]. It consists of two main components: a SIB that acts as a common, semantic-oriented store of information and device capabilities, and KPs, virtual and physical smart objects that interact with one another through the SIB. Various SIB implementations exist that conform to the M3 specification of which Smart-M3, developed by Nokia, was the first open source reference implementation released in 2009[3]. RDF Information Base System (RIBS), developed by VTT, is a C-based implementation of M3 targeted for devices with low processing power, but requires a large amount of memory [38].

10.5 ADK-SIB

The SIB implementation used during the second and the third design iteration is called ADK-SIB (Application Development Kit SIB) and was developed within the SOFIA project. The ADK-SIB is a Jena-based[4] SIB written in Java and runs on the OSGi framework.

Reasoning in the standard ADK-SIB is implemented using the Jena Ontology API, but only basic reasoning with symmetric properties and transitive properties is supported. Our main contribution to improve the ADK-SIB implementation was to implement support for OWL 2 RL/RDF Rules reasoning, as well as SPIN rules using the TopBraid SPIN API[5].

The ADK-SIB and SPIN API was first mentioned in Section 4.4.1.

When the SIB starts up, we first load the ontology, written in OWL 2, from a specified web address into our *asserted* model. We then load the OWL 2 RL specification, specified as SPIN

3 http://sourceforge.net/projects/smart-m3/
4 http://jena.sourceforge.net/
5 http://topbraid.org/spin/api/

rules, from another OWL file. We also load any custom SPIN functions into a third model. We then build a union model of the three models and store all the asserted triples in a hashmap to improve lookup efficiency. Finally the TopSPIN reasoning engine performs inferencing across the union model, and all the inferences are stored in the *inferred* model.

Whenever a new triple is added, removed or updated, the inferred model is cleared and inferencing is performed using the reasoning engine. This means that no inferencing needs to be performed when a query is run.

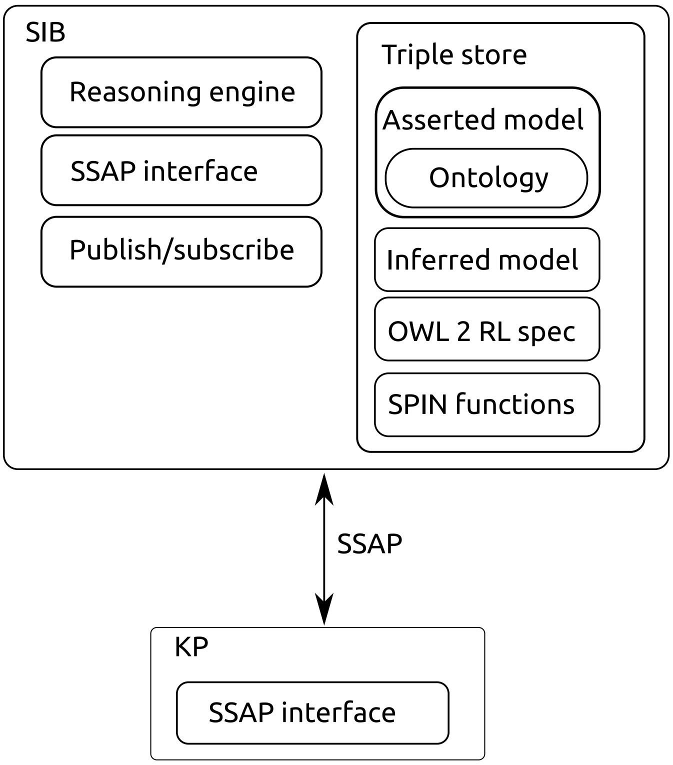


Figure 53: Our software architecture

The final software architecture is shown in Figure 53. The system performance of the software architecture was evaluated during the Smart Home pilot of the second design iteration, and this evaluation is described in the next chapter. A validation of the entire system, including the ontology, using the Cognitive Dimensions framework is also described.

11 EVALUATION

Statistics are a little like anarchists:
if you force them to stay in line, you're begging for trouble.

— Sarah Slobin [108], Graphics Editor, The Wall Street Journal

Parts of this chapter have previously appeared in [82].

In this chapter, two evaluations are described. First we will look at an evaluation of the system performance of the software architecture. This evaluation was performed during the smart home pilot of the second design iteration described in Chapter 4. Secondly, we will look at a method the author developed to evaluate ontologies based on the CD framework, as well as an evaluation of the ontology described in this thesis using the method.

The CD framework was first mentioned in Section 2.1.4.

11.1 EVALUATING THE SYSTEM PERFORMANCE

11.1.1 *Introduction*

To evaluate the software architecture described in Chapter 10, we compared it against a previous evaluation of the two M3-based smart space implementations, Smart-M3 and RIBS, described in Section 10.4. These implementations were evaluated by Eteläperä et al [38]. They performed both a qualitative evaluation and quantitative measurements. The performance measurements were made on a Intel Atom 1.6GHz laptop connected via a 100Mbps Ethernet router to a Intel Pentium M 1.7GHz laptop. The qualitative evaluation focused on documentation, installation process and portability as well as run-time usability. According to [38] RIBS is up to 237 times faster than Smart-M3 in certain instances, but it is reported that its memory model limits the number of use cases it can be applied to. RIBS uses static memory allocation with no disk storage and a bitcube triple store, which means that the maximum number of triples has to be known a priori.

Query time measurements for Smart-M3 indicated a query time of 4.4ms for one triple and 8.6ms for 10 triples. For RIBS a query time of 0.65ms was measured for one triple. RIBS did not support querying 10 triples at the time the evaluation was performed. Subscription time measurements indicated a subscription indication time of 140ms for Smart-M3, while RIBS measured 0.75ms.

Bhardwaj et al. [13] compared Smart-M3 against their Open Source Architecture for Sensors (OSAS) framework. They did a performance analysis based on end-to-end delay measurements between the smart objects in smart spaces. The analysis shows that the end-to-end delays are mostly dominated by KP-to-SIB updates, rather than the processing delays on KPs or on the SIB.

Luukkala et al. [70] used Smart-M3 with Answer Set Programming (ASP) techniques to handle resource allocation and conflict resolution. They used the SPICE Mobile Ontology[1] to describe device capabilities and ASP as a rule-based approach to reasoning. The SPICE ontology allows for the definition of device capabilities in a sub-ontology called DCS [121].

The SPICE DCS ontology was first mentioned in Section 7.2.3.

11.1.2 *Experimental setup*

The smart home pilot scenario was first described in Section 4.1.

In the smart home pilot, media content is shared among several devices in a smart home setting. Music is shared between a mobile device, a stereo speaker set and a lighting device that renders the mood of the music with coloured lighting. The music experience, consisting of both light and music information, is also shared remotely between friends living in separate homes through the lighting device. Other lighting sources, like the smart functional lighting and the smart wall wash lights are sensitive to user presence and the use of other lighting sources in the environment.

An overview of the smart home pilot is shown in Figure 20 on page 69, while a diagram showing the technical details is shown in Figure 21 on page 71.

The performance measurements were made in an environment that approximates a real-world home environment for these kinds of devices. Two wireless routers were placed in two different locations, bridged with an ethernet network cable. One router was configured to act as a DHCP server, while the other acted as a network bridge. The Connector KP, Music Player KP and SIB were connected to the router in location A, while the Sound/Light Transformer (SLT) KP was connected to the router in location B. All components were connected to the network via the 802.11g wireless protocol. The system specifications of each component used in the performance evaluation are shown in Table 6.

1 http://ontology.ist-spice.org/

COMPONENT	CPU	OS	MEMORY	LANGUAGE
SIB	Core 2 Duo 2.8GHz	Ubuntu 10.04	4GB	Java
SLT KP	Core 2 Duo 2.2GHz	Ubuntu 11.04	2GB	Java
Connector KP	Core 2 Duo 2.6GHz	OS X 10.6.8	4GB	Python
Music Player KP	ARM Cortex-A8	Maemo 5	256MB	Python
Presence KP	Pentium M	Ubuntu 10.04	512MB	Python
Lamp KP	Pentium M	Ubuntu 10.04	512MB	Python

Table 6: System specifications of components used in evaluation

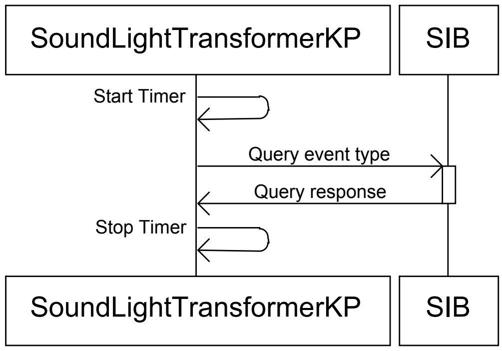


Figure 54: Sequence diagram of Sound/Light Transformer KP query measurement

Figure 54 and Figure 55 show the sequence diagrams of the measurements made for the SLT KP and the Connector KP respectively. During the pilot, 86 measurements were made by the SLT KP – each time an event was received. 961 measurements were made by the Connector KP – each time a user scans a tag. Note that the query name in 55 could differ between subsequent queries, as the user could be scanning a different tag every time.

For the music player KP, we measured the time between inserting a new event, and receiving an update from the SIB indicating that the specific event had occurred. First a subscription is made to the `PlayEvent` type, as shown in Figure 56. A new `PlayEvent` is generated by the KP, and when the KP is notified of this event by the SIB, the KP queries the SIB to determine if the notification is indeed for the event that it generated itself.

The Lamp-KP was connected to the decorative wall-wash lights (four LED lamps), creating coloured illumination on the wall of the room. The lamps are shown in Figure 58, including a description of its components. The Presence-KP determines the

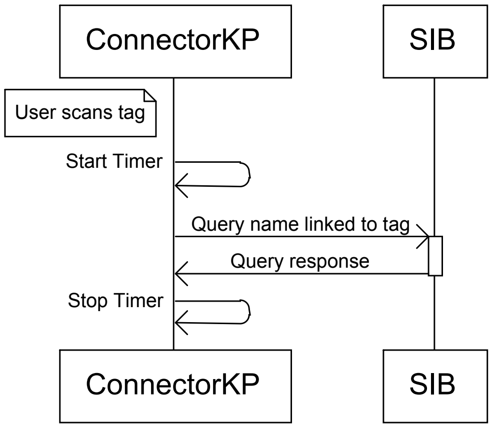


Figure 55: Sequence diagram of Connector KP query measurement

presence of a user in an activity area of a room and sends the presence information to the SIB. The Lamp-KP is subscribed to this presence information, and gets updated whenever the presence is updated by the Presence-KP to the SIB. There are two states to be updated by the Presence-KP on the SIB: `Away` and `Present`. Based on these states, the Lamp-KP turns the lamps on or off. For example, when the `Present` state is specified by the Presence-KP, the Lamp-KP sends the `ON` command to all lamps, and the `OFF` command when the `Away` state is specified. The Lamp-KP is also subscribed to the states of the SLT KP. The sequence diagram for the Presence-KP, SLT KP, Lamp-KP and SIB is shown in Figure 57.

11.1.2.1 *Reasoning setup*

For the pilot, constraint violation checking was disabled, as this introduced quite a large delay ($> 1000ms$), and was not necessary for the purposes of the pilot. Constraint checking ensures that instances in the triple store meet the constraints attached to classes and properties in an ontology. Constraint violation checks are computationally expensive and cannot be performed for each add, remove and update operation. One possible solution is to perform constraint violation checks at regular intervals and then remove the offending triples.

We made use of OWL 2 RL/RDF Rules in the smart home pilot, which is a *semantic* subset of OWL 2 *Full*. This should not be confused with the first part of the OWL 2 RL Profile[2], which is a *syntactic* subset of OWL 2 *DL*, and restricted in the type

2 http://www.w3.org/TR/owl-profiles/

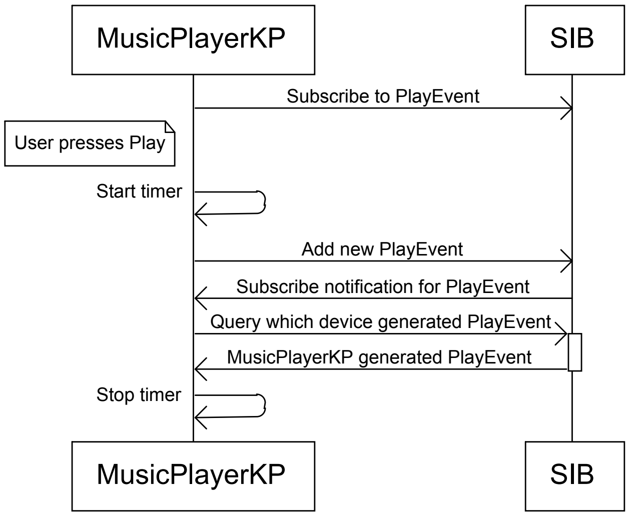


Figure 56: Sequence diagram of Music Player KP subscription measurement

of inferences that can be performed. In practice, most OWL 2 reasoners implement OWL 2 RL/RDF Rules (from here on known as OWL 2 RL). OWL 2 RL addresses a significant subset of OWL 2, including property chains and transitive properties. It is fully specified as a set of rules - in our case, as a set of SPIN rules. This means that it is even possible to select only the parts of OWL 2 that are required for a specific ontology, to allow for scalable reasoning.

During the smart home pilot, all SSAP messages received by the SIB were logged for further analysis.

SSAP is described in Section 10.3.

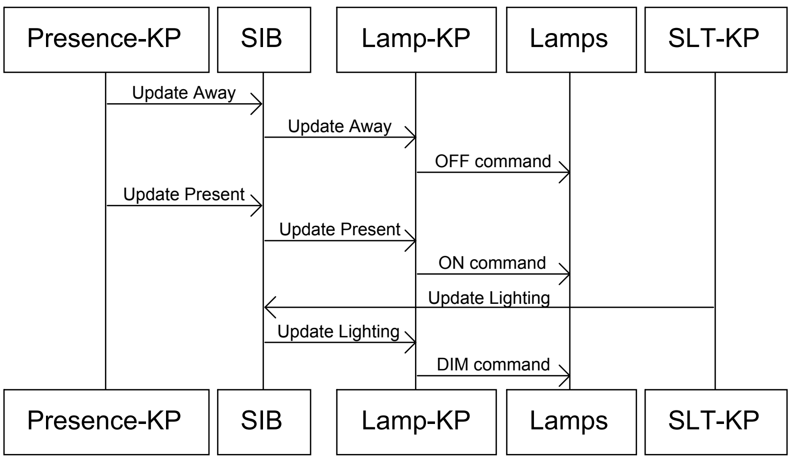


Figure 57: Sequence diagram of Presence-KP and Lamp-KP

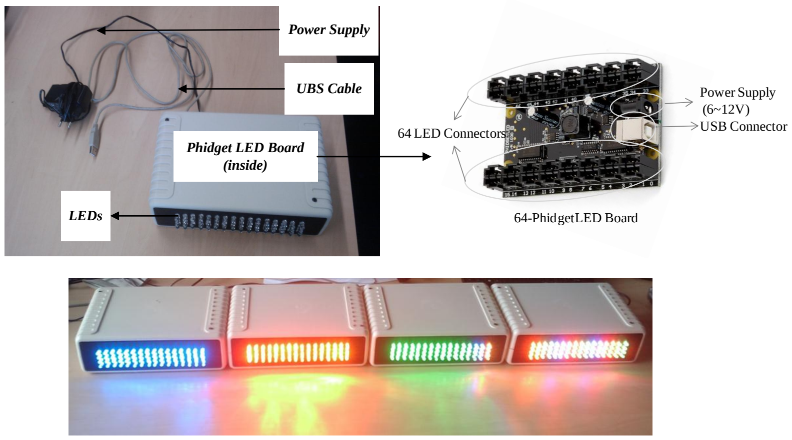


Figure 58: Lamp-KP

11.1.3 *Experimental Results*

After every reasoning cycle both the asserted and inferred models were written to disk, generating a total of 8306 models during the pilot. Reasoning was performed once, after all ontologies were loaded, and then for every add, remove and update operation. This resulted in a total of 5158 measurements of model size and reasoning time during the pilot. No reasoning was performed during queries.

During the pilot, 70655 total queries were performed by devices connected to the SIB. The time to perform each query was recorded on the SIB, and is shown in Figure 59. The histogram with bin size 25 is plotted on a logarithmic scale. Around 70000 queries take 2ms or less to complete, accounting for more than 99% of the queries. Of all the queries, only 3 queries took 30ms or longer to complete, with all queries completing in less than 60ms. Keep in mind that these measurements were performed on the SIB, hence the measurements do not take network latency into account.

Figure 60 shows the histograms, Gaussian Kernel Density Estimates (KDEs) and Cumulative Distribution Functions (CDFs) of the Connector KP and SLT KP *query time measurements*. A bin size of 20 and a bandwidth of 0.5 was used to plot the figures. It shows that the typical query time for the Connector KP is very short, with a few outliers that took a very long time to complete (35.2s). For the SLT KP, the case is similar, but there are no extreme outliers, with the longest query taking only 587ms to complete. Note that the KDE provides similar information

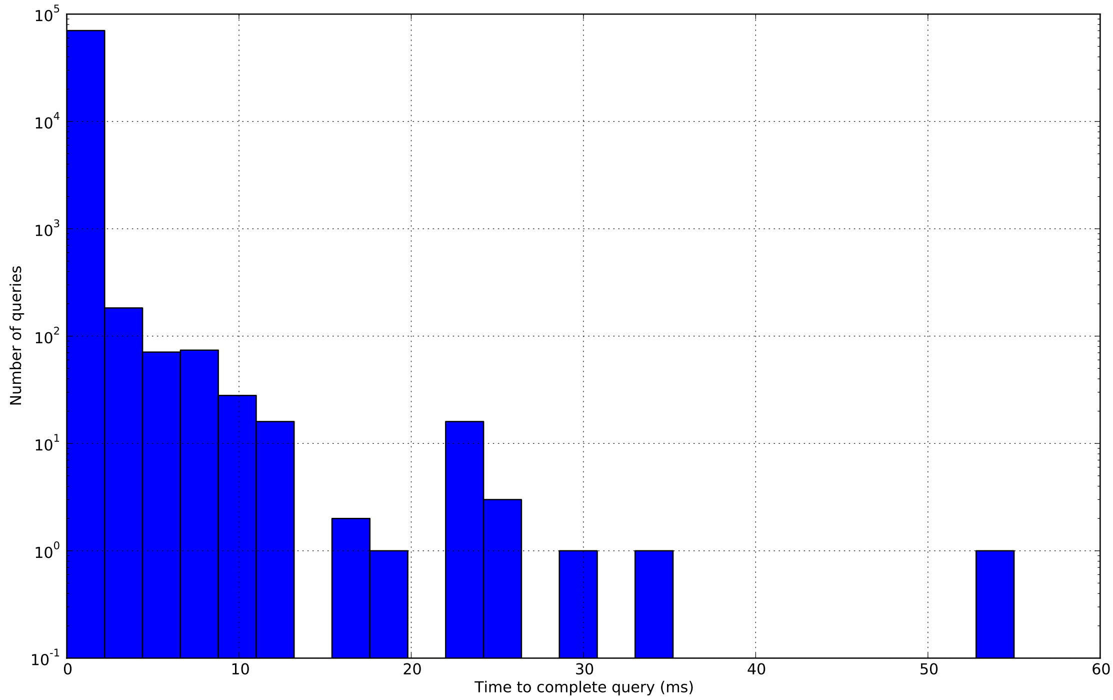


Figure 59: Query time measurements on SIB

to the histogram, but handles outliers more gracefully by not using binning, and also results in a smoother graph.

The CDF of the Connector KP indicates that queries taking more than two seconds to complete are very rare. The queries that do take longer than two seconds, take an unusually long time to complete. We believe that it could be related to problems in the wireless network, or related to the Python implementation of the Knowledge Processor Interface (KPI), as the problem did not present itself when using other KPI implementations.

For the SLT KP, most queries completed within 100ms, with very few queries taking longer than 500ms to complete.

For the music player KP, most subscription notifications completed in an average of 0.86s, as shown in Figure 61. Keep in mind that after the new PlayEvent is added, inferencing is performed on the triple store before the subscribe notification is generated. Summary statistics of Music Player KP, Connector KP and Sound/Light Transformer KP measurements are shown in Table 7.

In Figure 62, the following is shown:

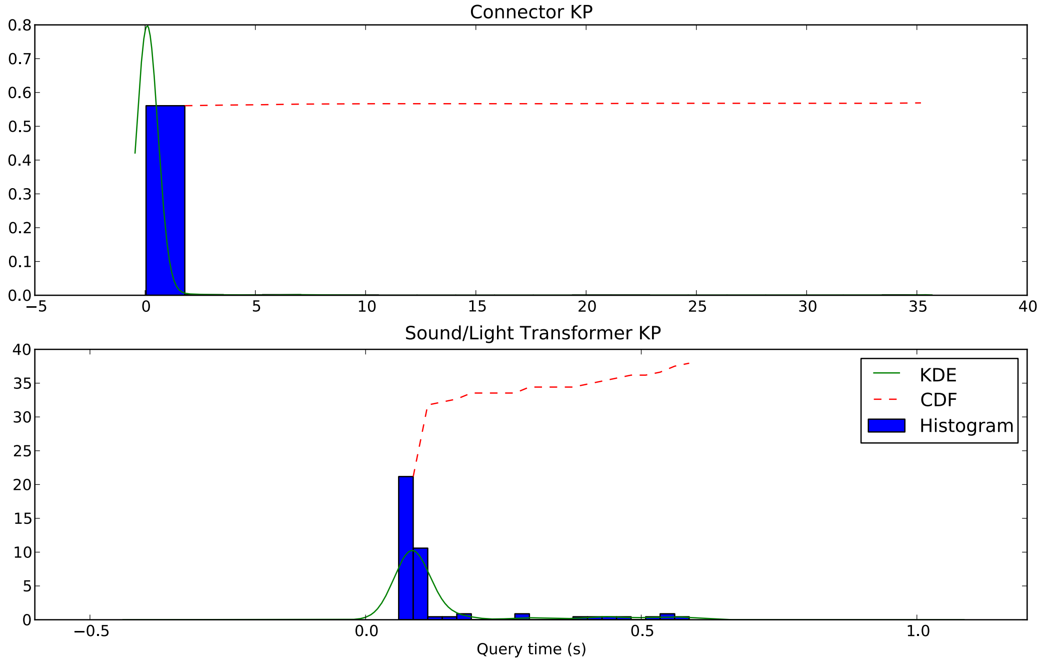


Figure 60: Histograms, kernel density estimates and cumulative distribution functions of Connector KP and Sound/Light Transformer KP measurements

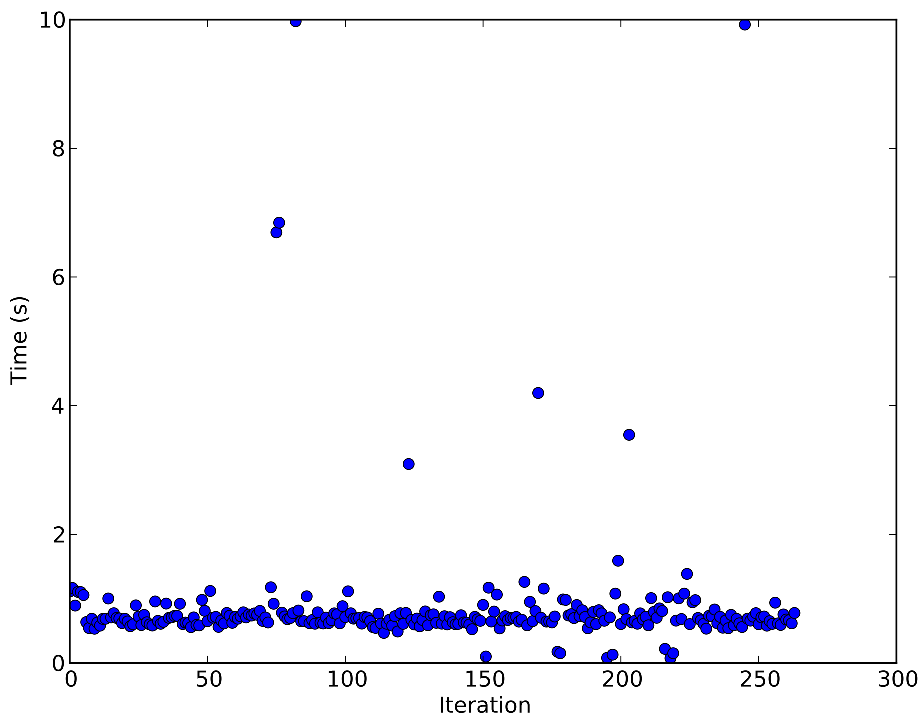


Figure 61: Subscription measurements of Music Player KP

COMPONENT	NR. OF OBS.	MIN. (S)	MAX. (S)	MEAN (S)	STD. DEV. (S)
Music Player KP	264	0.074	9.975	0.861	1.017
Connector KP	961	0.044	35.184	0.275	1.942
Sound/Light KP	86	0.06	0.587	0.131	0.122
Lamp-KP	98	0.012	0.049	0.03	0.006
Presence-KP	172	0.145	0.244	0.176	0.018

Table 7: Summary statistics of Music Player KP, Connector KP and Sound/Light Transformer KP measurements

COMPONENT	NR. OF OBS.	MIN.	MAX.	MEAN	STD. DEV.
Model size (nr of triples)	5158	1346	3396	2916.7	201.07
Inferred model size	5158	1369	1819	1501.8	107.6
Reasoning time (ms)	5158	181	2912	274.99	152.96

Table 8: Summary statistics for asserted and inferred model sizes and reasoning time, with model size indicated as number of triples

- Model size: Number of triples asserted by ontology or connected KPs
- Inferred model size: Number of triples inferred by reasoning engine
- Inferencing duration: Time (in ms) to complete one reasoning cycle

The sharp peaks indicate the times that the SIB was restarted. The first reasoning cycle after a restart takes about 3 seconds, with subsequent cycles taking on average 275ms (as seen in Table 8).

There is a slow but steady increase in the number of triples as new events get added to the triple store. After each restart these events are cleared and the base assertions loaded from the ontology. These assertions include the OWL 2 RL specification, stored as SPIN rules, which account for the large number of triples.

In Figure 63, the measurements show that the delay between the Presence-KP and the SIB is rather large with a considerable variance. The communication from the Presence-KP to the SIB

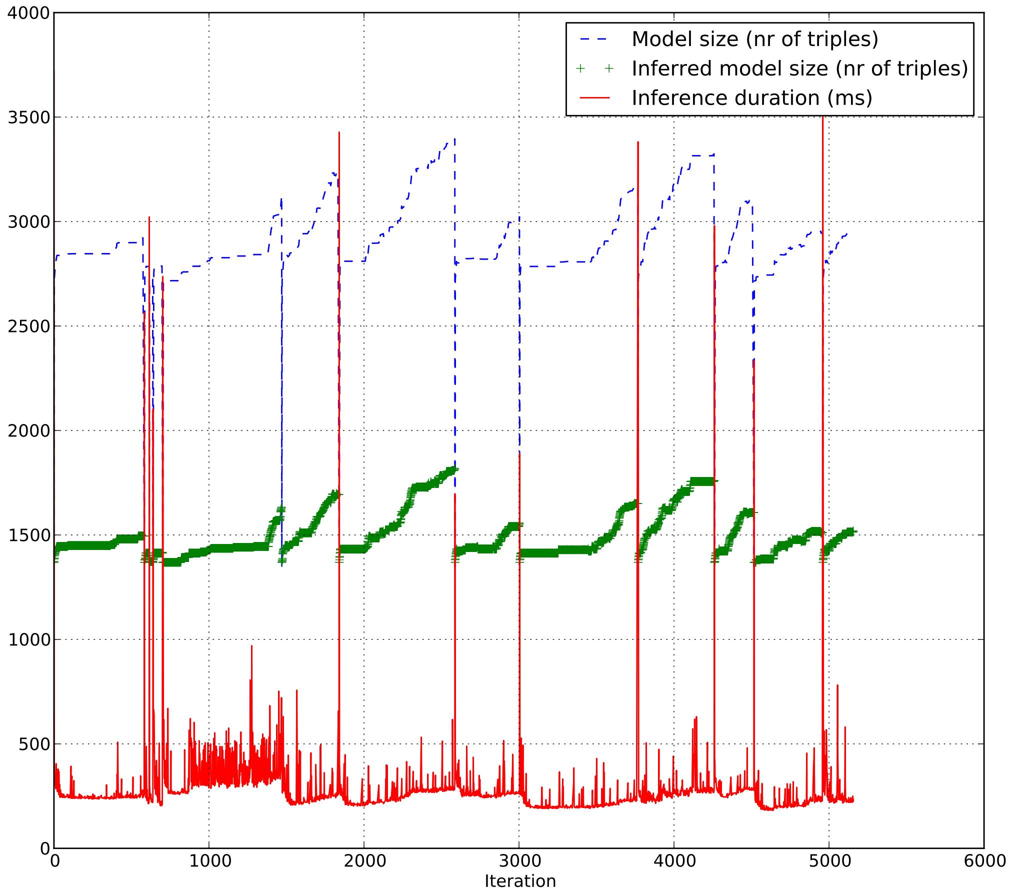


Figure 62: Size of asserted and inferred models for each iteration, including reasoning time

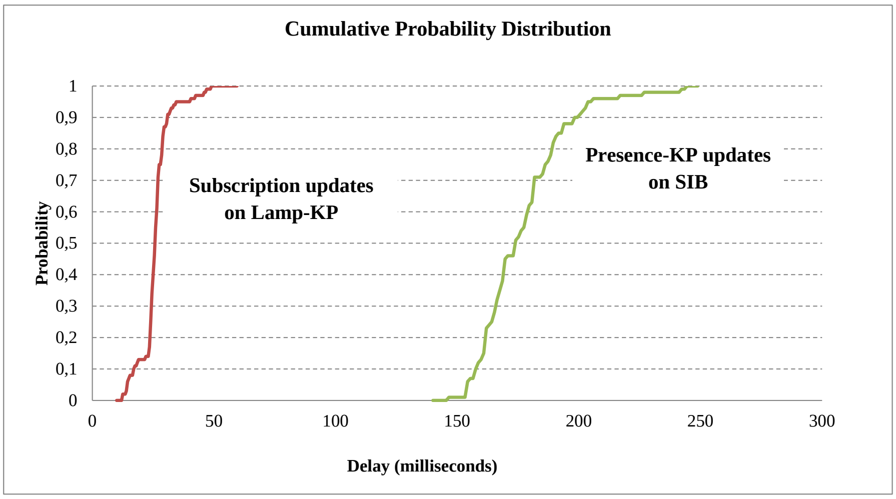


Figure 63: Cumulative probability distribution of delays between Presence-KP and SIB, as well as SIB and Lamp-KP

consists of an update request and the related confirmation response. The average delay of 176.71 milliseconds is the largest component of the total end-to-end delay between the links. The communication from the SIB to the Lamp-KP consists of an indication event from the SIB due to a subscription and results in a query request from the Lamp-KP, terminated by a confirmation response by the SIB. There is some variance in the communication delay and the average delay of 25.87 milliseconds might become problematic when the SIB has to inform and handle multiple subscribers.

11.1.4 *Discussion*

During the pilot, most problems could be attributed to problems with the wireless network. A number of devices from different manufacturers experienced intermittent problems while connected to the SIB. For the purposes of the pilot, these devices were then connected to the SIB via ethernet. This does, however, demonstrate some of the problems with existing wireless networking technologies. It cannot be expected that a device with a wireless connection will always stay connected to the smart space, even when it is within range of the wireless router.

If we compare our query time measurements to the ones performed on Smart-M3 (4.4ms) and RIBS (0.65ms), we can see that the KPs were substantially slower: The Connector KP at 44ms, the Music Player KP at 74ms and the Sound/Light KP at 60ms. This can be attributed to additional network latency in the field study that approximated a real-world environment. Query measurements that were performed on the ADK-SIB directly (0.445ms) shows that it performs even better than RIBS, but this is not directly comparable as the measurements do not include network latency time. If network latency is taken into account, measured at 0.43ms per packet round-trip by [38], RIBS is still faster.

The subscription indication measurements of our setup are also significantly slower. While the Smart-M3 measurement was only 140ms, the Music Player KP measures around 860ms. This is mostly due to the additional time required for reasoning, which on average takes about 275ms. Reasoning improves the flexibility and capabilities of the SIB to such an extent that it is worth the hit in performance. Other contributing factors include the number of devices used, the number of triples and the network environment. While the Smart-M3 and RIBS measurements were made in a lab environment, our ADK-SIB measurements were made in conditions that approximate a real-world environment, including having a larger number devices active at the same time.

In the 1960s there already existed some controversy over the maximum allowable response times in human-computer interfaces [75]. It was shown that different human actions will have different acceptable response times. While the delay between pressing a key and visual feedback should be no more than 0.1-0.3 seconds, the response to a inquiry request may take up to 2 seconds. The performance measurements are still well within this two-second limit, indicating that from a user's point of view, when performing routine tasks, the system is responsive enough. The user study and interviews performed during the pilot seem to confirm this, as no participant indicated any issues with regards to the responsiveness of the system.

The scalability of the system could still be evaluated with larger triple sizes, but we do not foresee any scalability issues, due to the platform and software architecture used.

Not only the software architecture described in this thesis should be evaluated, but also the ontology that was developed during the three design iterations. This is the focus of the next section.

11.2 EVALUATING THE ONTOLOGY

11.2.1 *Introduction*

In the book Beautiful Data [102], the notion of beauty is described as "a simple and elegant solution to some kind of problem". In a paper on the notion of beauty when building and and evaluating ontologies, D'Aquin and Gangemi [27] argue that the GoodRelations[3] e-commerce ontology could be considered an example of a beautiful ontology. GoodRelations is an OWL 1 DL ontology that is used by stores to describe products and their prices and features. Companies using the ontology include, Google, BestBuy, Sears, K-Mart and Yahoo. It addresses a complex domain and covers many of the complex situations that can occur in the domain. It is well designed, ontologically consistent, lightweight and used extensively by practitioners in the domain.

Hepp [55], creator of the GoodRelations ontology, describes a number of characteristics that can be used to evaluate an ontology:

- High expressiveness: An ontology of higher expressiveness would be richly axiomatised in higher order logic, while a simple vocabulary would be of low expressiveness.
- Large size of community: As ontologies are considered community contracts, an ontology targeted towards a large community should be easy to understand, well documented and of a reasonable size.

For more on community contracts, see Section 9.

- Small number of conceptual elements: A larger ontology is more difficult to visualise and review. A reasoner could also take a long time to converge when the ontology is very large.
- Low degree of subjectivity: This is very much related to the domain of the ontology, where something like religion would be more subjectively judged than engineering.
- Average size of specification per element: The number of axioms or attributes used to describe each concept influences the ontological commitment that must be made before adopting the ontology.

3 http://www.heppnetz.de/projects/goodrelations/

We consider our ontology to be of higher expressiveness compared to other ubiquitous computing ontologies. Most of the existing technologies try to describe a large number of concepts, while the number of actual axioms used to describe these concepts are rather low. The size of the community is small at the moment, as only project partners in the SOFIA project and students have been using the ontology up to now. Although the final version of our ontology contains 1192 asserted triples, the number of conceptual elements are low. There are only 34 OWL classes, 29 object properties, 7 datatype properties and one magic property, making the ontology easy to visualise and review. Based on our experiments as described in the previous, the ontology also converges in a reasonable amount of time.

We now look at a method we developed to evaluate ontologies using the CD framework, a framework that has been used to evaluate notational systems and programming environments [51], and has also been used to evaluate two of the related projects described in Section 2.1: AutoHAN [15] and e-Gadgets [73].

11.2.2 *Validating the work using Cognitive Dimensions*

The CD framework is a broad-brush approach to evaluating the usability of interactive devices and non-interactive notations, e.g. programming languages and APIs. It establishes a vocabulary of terms to describe the structure of an artefact and shows how these terms can be traded off against each other. These terms are, at least in principle, mutually orthogonal.

Traditional HCI evaluation techniques focus on 'simple tasks' like deleting a word in a text editor, or trying to determine the time required to perform a certain task. They are not well suited to evaluating programming environments or notational issues. The CD framework has been used to perform usability analyses of visual programming environments [51] as well as APIs [25]. Mavrommati et al. [73] used the framework to evaluate the usability of an editing tool that is used to manage device associations in a home environment.

Microsoft [25] used the CD framework to evaluate API usability, as part of a user-centred design approach to API design. Every API has a set of actions that it performs. However, developers browsing the API might not comprehend all the possible actions that the API offers. In a usability study they asked a group of developers to use an API to perform a set of tasks, and then asked a set of questions for each dimension. For example, for role expressiveness(see Section 11.2.2), the question was posed that when reading code that uses the API, if it was easy to tell what each section of the code does and why.

For our evaluation, we focused on a subset of cognitive dimensions that are related to ubiquitous computing ontologies and systems. What follows is a list of these dimensions, including a short description where necessary, as well as an example question.

LEVELS OF ABSTRACTION An abstraction is a grouping of elements that is treated as one entity, either for convenience or to change the conceptual structure [51]. *What are the minimum and maximum levels of abstractions?*

CLOSENESS OF MAPPING *How clearly did the available components map onto the problem domain?*

CONSISTENCY *When some part of the ontology has been learnt, how much of the rest can be inferred by the developer? Where there were concepts in the ontology that mean similar things, is the similarity clear from the way they appear?*

VISCOSITY To solve problems of viscosity, usually more abstractions (see earlier definition) are introduced in order to handle a number of components as one group, for example in object-oriented programming [51]. An example of viscosity is where it is necessary to make a global change by hand because the environment used does not have a global update tool. *How much effort was required to make a change to the environment?*

ROLE EXPRESSIVENESS The dimension of role expressiveness is intended to describe how easy it is determine what a certain part is for. *Are there parts of the ontology that are particularly difficult to interpret?*

HARD MENTAL OPERATIONS *Are there places where the developer needs to resort to fingers or pencilled annotation to keep track of what is happening? What kind of things required the most mental effort with this system and ontology?*

ERROR-PRONENESS *Did some kinds of mistakes seem particularly common or easy to make?*

HIDDEN DEPENDENCIES *If some parts were closely related to other parts, and changes to one may affect the other, are those dependencies visible?*

11.2.3 *Method*

The original CD questionnaire [14] was adapted for use with ubiquitous computing ontologies. Non-relevant questions were eliminated and some wording and questions were adjusted to the subject matter, without changing the fundamental meaning of the questions themselves.

As mentioned in Section 6.2.2, our academic partners in Bologna published a journal paper [9] based on an independent implementation of our work.

Developers of the SOFIA smart home pilot completed the questionnaire, as well as students and developers affiliated to the University of Bologna, for a total of 17 correspondents. The SOFIA developers used the ontology and software architecture for a couple of weeks in order to construct the smart home pilot. The students in Bologna took part in a course based on technology developed within the SOFIA project, where they also made use of the ontology and software architecture.

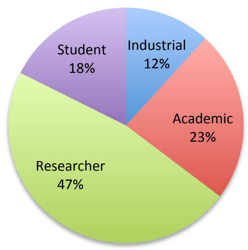
Three additional questions not directly related to the cognitive dimensions, but meant to elicit more general responses about the usability of the ontologies and the system, were included in the questionnaire.

11.2.4 *Results*

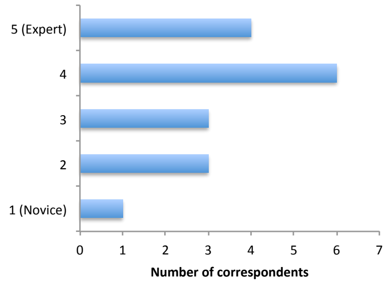
11.2.4.1 *Levels of abstraction*

> *Were you able to define your own concepts and terms using the system and ontology? Did you make use of different levels of abstraction?* An abstraction is a grouping of elements to be treated as one entity. In the ontology, these are defined as superclasses and subclasses, e.g. `PlayEvent` is a subclass of `MediaPlayerEvent`. Please indicate to what extent you made use of different levels of abstraction. If you did not use it, please indicate why.

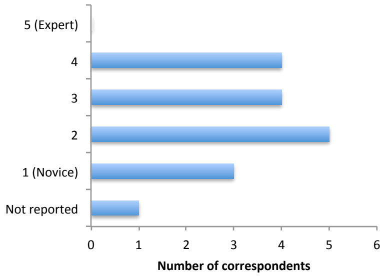
Most developers (14 out of 17) were able to make use of the existing concepts as defined, where the definition included different abstraction levels. Where necessary, developers (7 out of 17) were able to define their own concepts using different abstraction levels. One group used a simple ontology that did not



(a) Background of correspondents



(b) Correspondents' experience with software engineering and programming



(c) Correspondents' experience with ontologies and semantic technologies

Figure 64: Correspondent demographics

require different levels of abstraction. As the level of knowledge about ontologies differed between different parties working on the same project, this necessitated the simplification of the ontology to a schema without semantics in some cases. One group avoided ontological reasoning altogether by embedding the logic in the KP itself.

11.2.4.2 *Closeness of mapping*

> *How clearly did the available components map to the problem domain?* Did you have to define any of your own components, or were any special tricks needed to accomplish specific functionality?

Most of the developers (13 out of 17) experienced a clear, consistent mapping, with the domain mapped to already available components. In a few cases, developers developed their own components from scratch, or in addition to the available components (5 out of 17).

While it was easy to achieve the required functionality, it remains difficult to achieve component re-use. This becomes problematic for achieving emergent intelligent behaviour. It was also stated that more detailed descriptions of device capabilities, for example the coverage area of a presence sensor, are required.

11.2.4.3 *Consistency*

> *Where there were concepts in the ontology that mean similar things, is the similarity clear from the way they appear?* Are there places where some things ought to be similar, but the ontology defines them differently? Please give examples.

Most developers (9 out of 13 - 4 developers chose not to answer this question) thought that similar entities in the ontology were subclassed correctly. It was indicated that `owl:sameAs` may be useful to indicate that different terms with the same meaning are in fact the same thing.

Afterwards we realised that developers not well acquainted with ontologies found it difficult to understand the difference between declaring entities using `rdfs:subClassOf` and declaring them as individuals or instances, as well as how to model an entity that contains another entity.

Similar entities were not always instantiated in the same way, for example no state information was available for some smart objects. Where multiple domain ontologies with similar concepts were used, these concepts were not aligned - most developers expected some upper (or core) ontology to align and unify main concepts. Concepts need a clear textual description and usage examples to make them easier to understand.

11.2.4.4 *Viscosity*

> *How much effort was required to make a change to the environment? Why?* How difficult is it to make changes to your program, the ontology or the system? For example, was it necessary to make a global change by hand because no global update tools were available?

Although different domains used different terms to define ontological concepts, most developers (10 out of 17) found it quite easy to make changes for ontological agreement. However, changes to the ontology sometimes necessitated changes at code level. In most cases, it was easier to adapt to changes on a semantic level, as the KP domain boundaries were well defined.

Using ontologies made it easier to allow for definition changes at run-time. Depending on the inferencing method used, changes to the ontology could require some existing inferences to be removed.

One developer working with an embedded system found changes to be more difficult to implement, as it still required rebuilding images and downloading them to embedded boards for each modification. Another developer found it difficult to view changes made to the environment, due to a lack of tools to explore the contents of the SIB.

11.2.4.5 *Role expressiveness*

> *Are there parts of the ontology that are particularly difficult to interpret?* How easy is it to answer the question: 'What is this bit for?' Which parts are difficult to interpret?

Most responses (12 out of 17) indicated that the ontology was easy to understand. More clarifying comments inside the ontology could be useful - this can be implemented using the `rdfs:comment` field. One developer indicated that application ontologies (ontologies that are device-specific) were still hard to interpret.

Some concepts might be instinctively interpreted differently, but the defined meaning became clear when viewed in context with the rest of the ontology. The ontology provides the structure that is necessary to make sense of the concepts.

11.2.4.6 *Hard mental operations*

> *What kind of things required the most mental effort with this system and ontology?* Did some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?

5 of the 17 developers indicated that ontologies are not an easy concept to grasp and that common practice is not always clear. Once familiar with ontologies, and understanding the specific ontologies involved, developers thought they were easy to use.

There is still effort required to make the system adaptive. This issue is also mentioned in Section 11.2.4.2.

11.2.4.7 *Error-proneness*

> *Did some kinds of mistakes seem particularly common or easy to make? Which ones?* Did you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?

At least two developers found it quite easy to mistype string literals, indicating that it would be better to define entities to represent strings that occur more than once. Mistyping URIs was another common error, mentioned by two developers. One developer mentioned mixing up namespaces as an error that commonly occurred. Developers not familiar with ontologies also mixed up or misunderstood the differences between URIs and literals.

11.2.4.8 *Hidden dependencies*

> *If some parts were closely related to other parts, and changes to one may affect the other, are those dependencies visible?* What kinds of dependencies were hidden? How difficult was it to test the implemented system? Were there hidden faults that were difficult to find?

Three developers noted that using ontologies made the relationships among entities/parts more visible.

Changes to the ontology may affect others, therefore versioning and update notification are important. In the system used no errors were raised when components adhered to different versions of the ontology. Broken dependencies were only visible when the overall system failed.

One developer indicated that the publish/subscribe approach followed by the system architecture allowed for minimal dependencies due to loose coupling.

Three developers noted that it was difficult to determine when something did not work. This was especially noticeable in the case of subscriptions, were it becomes really difficult to understand why a certain subscription-based notification was not received.

One developer suggested that an ontology viewer could be used to make dependencies more visible, and that the Protégé ontology editor is too complex for most users.

The Protégé ontology editor was first mentioned in Section 4.6.

11.2.5 *Non-CD related questions*

Some questions in the questionnaire were not directly related to the cognitive dimensions, but were meant to elicit more general responses to the usability of the ontologies and system.

Question 1. *What obstacles made it difficult to use the system?*

This question was based on a survey done by [98] to determine aspects that make APIs hard to learn. The question was phrased as follows:

> *What obstacles made it difficult for you to use the system?* Obstacles can have to do with the system itself, with your background, with learning resources etc. List the three most important obstacles, in order of importance (1 being the biggest obstacle). Please be more specific than the general categories mentioned here.

Responses included:

- Stability, performance, network issues (6 responses)
- Lack of background in ontologies (4 responses)
- Difficult setup and installation procedure (4 responses)
- Poor documentation (3 responses)
- Lack of proper tools for viewing and exploring contents of SIB (2 responses)
- Insufficient code examples (1 response)
- SSAP poorly documented (1 response)
- Reliability of subscription mechanism, especially on wireless networks (1 response)
- QNames not supported on all platforms, requiring full URI to be specified (1 response)
- Ontology agreement (1 response)

SSAP is described in more detail in Section 10.3.

QNames enable full URIs to be substituted by short prefixes.

Question 2. *What did you appreciate most about the system and ontology?*

Responses included:

- Semantic interoperability between different devices, manufacturers and architectures (6 responses)
- Ontology usable in different domains and more complex scenarios (3 responses)
- Easy and quick to define new applications based on ontologies (3 responses)
- Decoupling of interaction between components, i.e. all communication through broker, but could be single point of failure (2 responses)
- Using semantic connections to connect devices (1 response)
- Ability to react to context changes through subscriptions (1 response)

One respondent had the following insight: "Once you have agreed on the ontologies and the KP's functionalities, you can focus on handling the various subscriptions and inserting the necessary triples. One does not have to focus on the communication protocols used or on the communication with the other components themselves."

Question 3. *Can you think of ways the design of the system and ontology can be improved?*

Responses included:

- Tools for ontology design, as currently external tools are required (2 responses)
- Authentication/security/locking (2 responses)
- SIB discovery (1 response)
- Agreement on ontological concepts to be used by a technical group (1 response)
- Better documentation (1 response)

- Self-description of UI concepts and component functionality (1 response)
- Hierarchic Smart Spaces (1 response)
- Locality/routing/separation of message buses (1 response)
- Forcing a programming paradigm like Object-Oriented Programming (OOP) influences semantic treatment (1 response)
- How to handle faulty smart objects (1 response)

These results were communicated by the author to the other project partners in the SOFIA project at a review meeting.

11.2.6 *Discussion & Conclusion*

Ontologies allow developers to create additional levels of abstraction when the existing abstractions are not sufficient. The bigger issue seems to be unfamiliarity with ontologies, with some developers going so far as to embed all logic in the code itself in order to avoid using ontologies, as mentioned in Section 11.2.4.1.

For the ontologies we defined, there seems to be clear mapping between objects in the domain and the ontological entities that they are mapped to. Adding additional components where necessary did not present any problems. One area that needs more attention is the extending the level of detail for device capability descriptions. Keep in mind, however, that too many low-level primitives create a cognitive barrier to programming [51]. It is not easy to deal with entities in the program domain that do not have corresponding entities in the problem domain. For example, having many ways to describe a presence sensor, when only one or two of these are relevant to the problem domain, makes it more difficult for the developer to comprehend.

When creating an ontology, it is important to provide clear textual descriptions, clarifying comments and usage examples for concepts to make them easier to understand. The GoodRelations e-commerce ontology is a good example of how this can be achieved.

GoodRelations was first discussed in Section 11.2.1.

Tools to explore the contents of the triple store more efficiently could decrease viscosity, as it would simplify viewing changes made to the environment and make dependencies more visible. Existing ontology viewers are still considered complex to use, usually only by ontology experts. Tools to automatically detect namespaces and prevent mistyping of URIs and strings used in the ontology would also be very useful.

Even though the Semantic Web community has been working on improving ontology standards for more than a decade, the main hurdle to adoption seems to be developers' unfamiliarity with ontologies. Education and more examples of successful implementations could help to alleviate this problem. The other issue that needs to be addressed is improving the tools used to create and view ontologies.

In addition to the strengths and limitations of the software architecture and the ontology described in this chapter, we discuss more general conclusions and achievements in the next chapter.

12 CONCLUSION

The holy grail of context awareness is
to divine or understand human intent.

— Anind Dey [29], ubicomp researcher

In this chapter we will discuss some of the results achieved by the work described in this thesis, and to what extent it validates the hypothesis and answers the research questions set out in Section 1.2.5. We will also discuss some of the lessons learned during the time spent working on this project.

Research work is no longer a one-man show. Apart from the close collaboration between Van der Vlist [116] and myself, we also had to work closely with the other partners in the SOFIA project. A large part of such a project is the work on technological integration, where the focus is on the interoperability of devices. This technological integration work was not always discussed in detail in the descriptions of the three design iterations, where the focus was mainly on showing our own contributions.

Intentional, incidental and expected interactions were introduced in Section 2.5.2.

One thing we learned from working in the ubiquitous computing domain is that automation should be used to simplify the complexities of technology, not necessarily to automate everything in the real world. It is much easier for a user to create a working mental model of his/her surroundings when explicitly interacting with things in the world. When incidental interactions occur and something happens automatically, there is a greater chance that the user will construct an incorrect mental model, and then expect a result that may be inconsistent with how the system actually works.

12.1 ACHIEVEMENTS AND OBSERVATIONS

The word serendipitous in the context of this thesis means that something is discovered by chance in a beneficial way, i.e. devices discover each others' functionality and make use of it.

At the beginning of the project we set out to create an ontology and software architecture that could enable serendipitous interoperability between devices in a smart environment. An ontology was created to model user interaction and devices in a smart environment consisting of multiple interactions and multiple devices.

The ontology and software architecture described in this thesis enable the creation of ensembles of devices. For example, the alarm functionality of a mobile phone, a wakeup service and a lamp can be combined to create an ensemble of devices with wakeup light functionality. This enables serendipitous interoperability, and we are excited to see what kinds of ensembles people come up with in future.

One side effect of enabling serendipitous interoperability in a smart environment is that there are now multiple ways to achieve the same goal. For example, if the user connects the alarm clock functionality on his/her mobile phone to the clock radio on his/her bedside table, the alarm can be set on either the phone or the clock radio – whichever way the user prefers. Van der Vlist's [116] observations indicate that this means that even though the users' mental models only partly matches the system they are interacting with, they are still able to achieve their goals.

A method to evaluate the usability of ontologies and systems for developers of smart environments was developed, based on the CD framework. While the evaluation method does not allow us to assign quantitive measures for usability, it does provide a vocabulary and framework for discussing usability issues with ontologies and smart environments.

The ontology and software architecture described in this thesis has proven to be easier to use and more flexible than existing methods, like storing device and service descriptions in a relational database. Developers were able to make use of existing ontology concepts as defined, as well as define their own concepts where necessary. Once familiar with how ontologies work, they found the ontology easy to understand. Ontologies also made the relationships between various entities more visible.

The system performance of the software architecture compares favourably to similar systems, as described in Section 12.3 below. Semantic reasoning was also shown to be a viable alternative to other approaches, like verification modelling languages and model checkers, also described in more detail in Section 12.3 below.

We now discuss some of the results of the work in more detail.

12.2 PROVIDING AFFORDANCES AND FEEDBACK FOR SMART OBJECTS

In a GUI, there are six fundamental interaction tasks, as described in Foley's seminal paper [42] In contrast, there are numerous activities that can be performed with or on a physical object, for example squeeze, tap and push. There are also no standard input/output devices, for example movement may be measured with an accelerometer, camera or infrared sensor. A user action, within a given interaction, may be distributed across multiple physical objects, as there is no single point of interaction [34].

Foley's taxonomy was discussed in Section 2.4.

Addressing a system with a GUI is very clear: the user uses an input device attached to the system. In a smart environment it is not always clear which devices form part of the system. In most systems using tangible interfaces, devices are augmented with RFID tags or IR transmitters, where they can be scanned or pointed at to initiate communication. If these tags and sensors are attached unobtrusively to devices, it is difficult for users to distinguish which devices form part of the smart environment, as there are no visual affordances.

Tangible interfaces were discussed in Section 2.3.1.

Our approach to solving this problem was to make extensive use of feedback and feedforward. For example, we use augmented feedforward to display a device's functional possibilities at the time a connection between two devices is being made. We also use feedback to confirm user actions, using augmented feedback where direct functional feedback is not available.

Feedback and feedforward is discussed in more detail in Section 5.4.1.

12.3 SOFTWARE ARCHITECTURE

Even with all the different toolkits and systems to demonstrate the usefulness of ubiquitous computing technology, as described in Section 2.1, building these kind of systems is still a complex and time-consuming task due to a lack of appropriate infrastructure or middleware-level support [53].

Chapter 10 in this thesis can act as a reference design for future implementations. From the work described in this thesis we have shown that having an architecture based on the blackboard pattern and publish/subscribe paradigm works well. We evaluated the suitability of such a combination to handle ontology-based ubiquitous computing environments, with promising results.

The ØMQ protocol, mentioned in Section 10.3, can run with or without a dedicated message broker.

However, having a centralised information broker is only one solution. Hybrid approaches using both centralised and decentralised techniques should also be explored.

The EventHeap system, first mentioned in Section 2.1.2, also used a tuple space protocol to establish communication between devices. A performance evaluation on a Pentium II 450 MHz with 256 MB of RAM shows that the system can provide latency below 100 ms for 12 different applications each generating and receiving 10 events per second [63]. Our query time measurements in Section 11.1.3 were also all below 100 ms.

Johanson and Fox [63] noted that both latency and scalability in an interactive environment are bounded by social constraints and human factors. Scalability is bounded by the number of people and devices interacting with one another to solve some problem, while latency on a local subnet is usually fast enough to be imperceptible to humans.

For subscription measurements, however, where time is also required for reasoning, our system performs slower. While an independent evaluation of the Smart-M3 platform showed a subscription measurement time of 140 ms [38], our implementation takes an average of about 275 ms. As already discussed in Section 11.1.4, the reasoning improves the flexibility and capabilities of the system to such an extent that it is worth the hit in performance.

12.4 ONTOLOGIES

Most systems use programming language objects to represent knowledge about their environment. Because these representations require an a priori agreement on how they will be implemented in a system, they do not facilitate knowledge sharing in an open and dynamic environment [24]. Based on our work, we believe strongly that using ontologies to describe and reason about smart environments have a lot of potential.

Semantic Web technologies are well suited to ubiquitous computing scenarios. They have been designed to work at Web scale, they provide interoperability between heterogeneous data sources, and they rely on existing Web standards which allow for easy adoption [100].

The model checker is called SPIN (Simple Promela Interpreter), with no relation to the SPIN rules described elsewhere in this thesis.

One alternative to semantic reasoning and ontologies is to use verification modelling languages and model checkers. Calder et al [21] used the Promela language to represent the ruleset of an event-driven, context-aware homecare activity monitor system. Promela is a C-Like state-based language for modelling communicating, concurrent processes. A model checker was used to perform redundancy checking. Verification times for a ruleset of eight rules varied between 12 and 34 minutes. A specialised SAT[1] solver only took around 15 ms to solve each instance as a SAT model, but converting the rule set to SAT models and then performing redundancy checking takes around 5 seconds.

Given that the verification modelling language approach is geared towards a full state space exploration, the results are not directly comparable to ours. However, considering what both these approaches are trying to achieve, it is still an interesting comparison. Our average reasoning time of 275 ms for a ruleset of similar size indicates that using ontologies is a viable approach for ubiquitous computing scenarios. It should be noted that they were investigating using semantics and ontologies to perform verification as part of their future work.

Reasoning times are shown in the experimental results in Section 11.1.3.

12.5 LOW COST, HIGH TECH

One area that we focused on in this project was to see what kind of low-cost products are available that have similar functionality to more expensive equipment. For example, while some 13.56MHz RFID readers currently retail for thousands of euros, the Touchatag reader we used retails for around €30.

The Touchatag reader was first discussed in Section 3.3.1.

The Nokia 5800 XpressMusic phone we used in the first design iteration provides a touch screen, WiFi and Bluetooth connectivity and accelerometer for a fraction of the cost of other phones with similar functionality. Of course, there are some tradeoffs that need to be made, for example less processing power or slower responsiveness compared to more expensive models. We see this approach in a similar light to using rapid prototyping techniques, like paper or video prototypes, were tradeoffs are required in order to test out your ideas.

Another example is the Squeezebox radio used in our third design iteration, which currently sells for around €120. Comparable state-of-the-art wireless media systems cost upwards of €300 per device.

In their book on mobile interaction design, Jones and Marsden [64] describe the effect of ubiquitous computing with mobile devices in developing countries, using cheap and simple

1 SAT is an abbreviation for the Boolean satisfiability problem in computer science.

Greenfield mentioned the issue of second class technology during a summer school panel attended by the author.

technologies. For example, while Internet penetration in South Africa was only 7.1% of the population by 2005, mobile penetration was around 50%. We believe there is a lot of work to be done in this area. However, there is a note of caution from Adam Greenfield, author of the book Everyware [52], where he is worried about the creation of second class technology, for example where translating financial web applications into an SMS-based system.

12.6 FUTURE WORK

The ontologies developed as part of the work described in this thesis are considered a good starting point for future work. While they provide a good foundation for modelling device capabilities and interaction events, they cannot be considered comprehensive yet, due to the sheer number of devices out there. However, being written in OWL 2 and SPIN means that they can be easily extended.

Concepts introduced in the thesis, like semantic transformers and interaction primitives, can be improved upon. Interaction primitives do not yet fully describe all the possible user interaction capabilities of smart objects, while the features of semantic transformers can also be extended. The software architecture and ontologies were deployed and evaluated in the smart home environment, but could also be applied to other environments, like smart personal spaces or the smart city.

One aspect that we would like to explore in the future is to study the possibilities of minimising information overload in more detail. It should be possible to adapt the amount of information in the environment to your mood, for example whether you want to increase your performance, or just want to focus on relaxation and enjoyment.

`n3pygments` was used to perform syntax highlighting for the ontology and SPARQL fragments in this thesis.

There is room for improvement when it comes to documentation tools for ontologies. For example, there exist many tools that enable the automatic generation of API documentation from source code, like Doxygen[2] and Sphinx[3]. At the time of writing, there is only a limited set of documentation generators available for ontologies, for example OWLDoc[4]. An effort by the author to improve syntax highlighting for SPARQL and Turtle

2 http://doxygen.org
3 http://sphinx.pocoo.org
4 http://code.google.com/p/co-ode-owl-plugins/wiki/OWLDoc

syntax, called `n3pygments`, is available as open source[5] and was developed during this project.

The smart home pilot was discussed in Section 4.1.

Another aspect that was considered out of scope for the project was social awareness, for example notifying a close friend performing a similar activity or having a similar goal. We touched on this aspect during the smart home pilot, where music and lighting patterns could be shared between friends in two different locations. Combining research on social networks with the work described in this thesis could be an interesting direction for future research.

5 https://github.com/gniezen/n3pygments

Part IV

APPENDIX

APPENDIX

In this appendix the final version of the ontology is given in Turtle format. It is also available to download from:

`https://github.com/gniezen/ontologies/`

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <http://semantic.gerritniezen.com/ontologies/UserInteractionEvents.owl#> .
@prefix afn: <http://jena.hpl.hp.com/ARQ/function#> .
@prefix spin: <http://spinrdf.org/spin#> .
@prefix sp: <http://spinrdf.org/sp#> .
@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 spl: <http://spinrdf.org/spl#> .

<http://semantic.gerritniezen.com/ontologies/UserInteractionEvents.owl>
    spin:imports <http://topbraid.org/spin/owlrl-all> ;
    a owl:Ontology ;
    owl:imports <http://spinrdf.org/spin> ;
    owl:versionInfo "Created with TopBraid Composer"^^xsd:string .

:AdjustLevel
    a :Functionality .
:AdjustLevelEvent
    a owl:Class ;
    rdfs:subClassOf :InteractionEvent .
:Alarm
    a :Functionality .
:AlarmAlertEvent
    a owl:Class ;
    rdfs:subClassOf :AlarmEvent .
:AlarmEndEvent
    a owl:Class ;
    rdfs:subClassOf :AlarmEvent .
:AlarmEvent
    a owl:Class ;
    rdfs:subClassOf :InteractionEvent .
:AlarmRemoveEvent
    a owl:Class ;
    rdfs:subClassOf :AlarmEvent .
:AlarmSetEvent
    a owl:Class ;
    rdfs:subClassOf :AlarmEvent, :SetEvent ;
    owl:equivalentClass [
        a owl:Class ;
        owl:intersectionOf (:SetEvent
```

```
                [
                    a owl:Restriction ;
                    owl:allValuesFrom xsd:dateTime ;
                    owl:onProperty :dataValue
                ]
                [
                    a owl:Restriction ;
                    owl:cardinality "1"^^xsd:nonNegativeInteger ;
                    owl:onProperty :dataValue
                ]
            )
        ] .
:Audio
    a :MediaType .
:Binary
    a :RangeMeasure ;
    rdfs:comment "True/False, 0 or 1"^^xsd:string .
:Bridge
    a owl:Class ;
    rdfs:subClassOf :SmartObject ;
    owl:equivalentClass [
        a owl:Class ;
        owl:intersectionOf (:SmartObject
            :Sink
            :Source
        )
    ] .
:CueEvent
    a owl:Class ;
    rdfs:subClassOf :MediaPlayerEvent ;
    owl:disjointWith :PlayEvent, :StopEvent .
:Double
    a :RangeMeasure ;
    rdfs:comment "Capable of representing values from 0-99 (double digits)" .
:Event
    a owl:Class ;
    rdfs:subClassOf owl:Thing .
:FeedbackEvent
    a owl:Class ;
    rdfs:subClassOf :PreviewEvent .
:Functionality
    a owl:Class ;
    rdfs:subClassOf owl:Thing .
:IDType
    a owl:Class ;
    rdfs:subClassOf owl:Thing .
:IPAddress
    a :IDType .
:Identification
    a owl:Class ;
    rdfs:subClassOf owl:Thing .
:IncreaseLevelEvent
```

```
    a owl:Class ;
    rdfs:subClassOf :AdjustLevelEvent ;
    owl:equivalentClass [
        a owl:Class ;
        owl:intersectionOf (:AdjustLevelEvent
            [
                a owl:Restriction ;
                owl:onProperty :duration ;
                owl:someValuesFrom xsd:integer
            ]
        )
    ] .
:IndicatorEvent
    a owl:Class ;
    rdfs:subClassOf :PreviewEvent .
:InteractionEvent
    a owl:Class ;
    rdfs:subClassOf :Event .
:InteractionPrimitive
    a owl:Class ;
    rdfs:comment "Smallest addressable interaction element
that has a meaningful relation to the interaction itself"^^xsd:string ;
    rdfs:subClassOf owl:Thing .
:MediaPath
    a owl:Class ;
    rdfs:subClassOf owl:Thing .
:MediaPlayerEvent
    a owl:Class ;
    rdfs:subClassOf :InteractionEvent .
:MediaType
    a owl:Class ;
    rdfs:subClassOf owl:Thing .
:Music
    a :Functionality .
:PlayEvent
    a owl:Class ;
    rdfs:subClassOf :MediaPlayerEvent ;
    owl:disjointWith :CueEvent, :StopEvent .
:PreviewEvent
    a owl:Class ;
    rdfs:subClassOf :InteractionEvent .
:RFID_Mifare
    a :IDType .
:RGBValues
    a :MediaType .
:RangeMeasure
    a owl:Class ;
    rdfs:subClassOf owl:Thing .
:Real
    a :RangeMeasure ;
    rdfs:comment "Capable of representing more than 1000 discrete values (infinite)" .
:SemanticTransformer
```

```
spin:rule [
    sp:templates ([
            sp:object spin:_this ;
            sp:predicate :connectedTo ;
            sp:subject _:A2
        ]
        [
            sp:object _:A3 ;
            sp:predicate :connectedTo ;
            sp:subject spin:_this
        ]
    ) ;
    sp:where ([
            sp:object spin:_this ;
            sp:predicate :canConnectTo ;
            sp:subject _:A2
        ]
        [
            sp:object _:A3 ;
            sp:predicate :canConnectTo ;
            sp:subject spin:_this
        ]
        [
            sp:object _:A3 ;
            sp:predicate :connectedTo ;
            sp:subject _:A2
        ]
    ) ;
    a sp:Construct
], [
    sp:templates ([
            sp:object _:A1 ;
            sp:predicate spin:_this ;
            sp:subject _:A0
        ]
    ) ;
    sp:where ([
            sp:object _:A1 ;
            sp:predicate :canConnectTo ;
            sp:subject _:A0
        ]
    ) ;
    a sp:Construct
] ;
a owl:Class ;
rdfs:subClassOf owl:Thing ;
owl:disjointWith :SmartObject ;
owl:equivalentClass [
    a owl:Class ;
    owl:intersectionOf (:SemanticTransformer
        [
            a owl:Restriction ;
```

```
                owl:cardinality "1"^^xsd:nonNegativeInteger ;
                owl:onProperty :functionalitySource
            ]
        )
    ], [
        a owl:Class ;
        owl:intersectionOf ([
                a owl:Restriction ;
                owl:onProperty :canAcceptMediaTypeFrom ;
                owl:someValuesFrom :SmartObject
            ]
            [
                a owl:Restriction ;
                owl:onProperty :convertsMediaType ;
                owl:someValuesFrom :SmartObject
            ]
        )
    ] .
:SetEvent
    a owl:Class ;
    rdfs:subClassOf :InteractionEvent .
:Single
    a :RangeMeasure ;
    rdfs:comment "Capable of representing values from 0-9 (single digits)" .
:Sink
    a owl:Class ;
    rdfs:subClassOf :SmartObject ;
    owl:equivalentClass [
        a owl:Class ;
        owl:intersectionOf (:SmartObject
            [
                a owl:Restriction ;
                owl:onProperty :functionalitySink ;
                owl:someValuesFrom :Functionality
            ]
        )
    ] .
:SmartObject
    spin:rule [
        sp:templates ([
                sp:object _:A5 ;
                sp:predicate _:A4 ;
                sp:subject spin:_this
            ]
        ) ;
        sp:where ([
                sp:object _:A4 ;
                sp:predicate :functionalitySource ;
                sp:subject spin:_this
            ]
            [
                sp:object _:A4 ;
```

```
                sp:predicate :functionalitySink ;
                sp:subject _:A5
            ]
        ) ;
        a sp:Construct
    ], [
        sp:templates ([
                sp:object _:A6 ;
                sp:predicate :hasLastEvent ;
                sp:subject spin:_this
            ]
        ) ;
        sp:where ([
                sp:expression [
                    sp:arg1 spin:_this ;
                    a :getMaxDateRsc
                ] ;
                sp:variable _:A6 ;
                a sp:Bind
            ]
        ) ;
        a sp:Construct
    ], [
        sp:templates ([
                sp:object _:A8 ;
                sp:predicate :hasRFIDTag ;
                sp:subject spin:_this
            ]
        ) ;
        sp:where ([
                sp:object :RFID_Mifare ;
                sp:predicate :ofIDType ;
                sp:subject _:A7
            ]
            [
                sp:object _:A7 ;
                sp:predicate :hasIdentification ;
                sp:subject spin:_this
            ]
            [
                sp:object _:A8 ;
                sp:predicate :idValue ;
                sp:subject _:A7
            ]
        ) ;
        a sp:Construct
    ] ;
    a owl:Class ;
    rdfs:subClassOf owl:Thing .
:Source
    a owl:Class ;
    rdfs:subClassOf :SmartObject ;
```

```
    owl:equivalentClass [
        a owl:Class ;
        owl:intersectionOf (:SmartObject
            [
                a owl:Restriction ;
                owl:onProperty :functionalitySource ;
                owl:someValuesFrom :Functionality
            ]
        )
    ] .
:StopEvent
    a owl:Class ;
    rdfs:subClassOf :MediaPlayerEvent ;
    owl:disjointWith :CueEvent, :PlayEvent .
:SystemEvent
    a owl:Class ;
    rdfs:subClassOf :Event .
:TCPIPObject
    a owl:Class ;
    rdfs:subClassOf :SmartObject ;
    owl:equivalentClass [
        a owl:Restriction ;
        owl:hasValue :IPAddress ;
        owl:onProperty :hasIDType
    ], [
        a owl:Restriction ;
        owl:hasValue true ;
        owl:onProperty :communicatesByTCPIP
    ] .
:TimeSetEvent
    a owl:Class ;
    rdfs:subClassOf :SystemEvent .
:Triple
    a :RangeMeasure ;
    rdfs:comment "Capable of representing values from 0-999 (triple digits)" .
:acceptsMediaType
    a owl:ObjectProperty .
:canAcceptMediaTypeFrom
    a owl:ObjectProperty ;
    owl:inverseOf :convertsMediaType .
:canConnectTo
    a owl:IrreflexiveProperty, owl:ObjectProperty ;
    owl:propertyChainAxiom (:functionalitySource
        :functionalitySink
    ) .
:canIndirectlyConnectTo
    a owl:ObjectProperty ;
    rdfs:subPropertyOf :canConnectTo ;
    owl:propertyChainAxiom (:canConnectTo
        :canConnectTo
    ) .
:communicatesByTCPIP
```

```
    a owl:DatatypeProperty ;
    rdfs:domain :SmartObject ;
    rdfs:range xsd:boolean .
:connectedFrom
    a owl:ObjectProperty ;
    owl:inverseOf :connectedTo .
:connectedTo
    a owl:IrreflexiveProperty, owl:ObjectProperty ;
    rdfs:domain :Source ;
    rdfs:range :Sink .
:convertsMediaType
    a owl:IrreflexiveProperty, owl:ObjectProperty ;
    owl:propertyChainAxiom (:transmitsMediaType
        :isAcceptedMediaTypeOf
    ) .
:cueAt
    a owl:DatatypeProperty ;
    rdfs:comment "Cue at time (in milliseconds)"^^xsd:string ;
    rdfs:domain :CueEvent ;
    rdfs:range xsd:integer .
:currentDateTime
    spin:body [
        sp:resultVariables (_:A9
        ) ;
        sp:where ([
                sp:expression [
                    a afn:now
                ] ;
                sp:variable _:A9 ;
                a sp:Bind
            ]
        ) ;
        a sp:Select
    ] ;
    a spin:MagicProperty ;
    rdfs:subClassOf spin:MagicProperties .
:dataValue
    a owl:DatatypeProperty ;
    rdfs:range xsd:anySimpleType .
:duration
    a owl:DatatypeProperty .
:functionalitySink
    a owl:ObjectProperty .
:functionalitySource
    a owl:ObjectProperty .
:generatedBy
    a owl:ObjectProperty ;
    rdfs:range :SmartObject .
:getMaxDateRsc
    spin:body [
        sp:limit "1"^^xsd:long ;
        sp:orderBy ([
```

```
                sp:expression _:A11 ;
                a sp:Desc
            ]
        ) ;
        sp:resultVariables (_:A10
        ) ;
        sp:where ([
                sp:object spin:_arg1 ;
                sp:predicate :generatedBy ;
                sp:subject _:A10
            ]
            [
                sp:object _:A11 ;
                sp:predicate :inXSDDateTime ;
                sp:subject _:A10
            ]
        ) ;
        a sp:Select
    ] ;
    spin:constraint [
        spl:predicate sp:arg1 ;
        a spl:Argument ;
        rdfs:comment "Smart object that generated the interaction event"^^xsd:string
    ] ;
    a spin:Function ;
    rdfs:subClassOf spin:Functions .
:hasIDType
    a owl:ObjectProperty ;
    owl:propertyChainAxiom (:hasIdentification
        :ofIDType
    ) .
:hasIdentification
    a owl:ObjectProperty ;
    rdfs:range :Identification .
:hasInteractionPrimitive
    a owl:ObjectProperty .
:hasLastEvent
    a owl:ObjectProperty .
:hasMediaPath
    a owl:ObjectProperty .
:hasRFIDTag
    a owl:DatatypeProperty .
:hasRangeMeasure
    a owl:ObjectProperty ;
    rdfs:domain :InteractionPrimitive ;
    rdfs:range :RangeMeasure .
:idValue
    a owl:DatatypeProperty ;
    rdfs:domain :Identification .
:identificationOf
    a owl:ObjectProperty ;
    owl:inverseOf :hasIdentification .
```

```
:inXSDDateTime
    a owl:DatatypeProperty ;
    rdfs:domain :Event ;
    rdfs:range xsd:dateTime .
:indirectlyConnectedTo
    a owl:ObjectProperty ;
    rdfs:subPropertyOf :connectedTo .
:isAcceptedMediaTypeOf
    a owl:ObjectProperty ;
    owl:inverseOf :acceptsMediaType .
:isFunctionalityOfSink
    a owl:ObjectProperty ;
    owl:inverseOf :functionalitySink .
:isFunctionalityOfSource
    a owl:ObjectProperty ;
    owl:inverseOf :functionalitySource .
:isInteractionPrimitiveOf
    a owl:ObjectProperty ;
    owl:inverseOf :hasInteractionPrimitive .
:launchesEvent
    a owl:ObjectProperty ;
    owl:inverseOf :generatedBy .
:mediaOriginator
    a owl:ObjectProperty .
:mediaPathOf
    a owl:ObjectProperty ;
    owl:inverseOf :hasMediaPath .
:mediaSourceSO
    a owl:ObjectProperty .
:ofIDType
    a owl:ObjectProperty ;
    rdfs:domain :Identification ;
    rdfs:range :IDType .
:tempConnectedTo
    a owl:ObjectProperty .
:transmitsMediaType
    a owl:ObjectProperty .
_:A0
    sp:varName "source"^^xsd:string .
_:A1
    sp:varName "sink"^^xsd:string .
_:A10
    sp:varName "lastEvent"^^xsd:string .
_:A11
    sp:varName "last"^^xsd:string .
_:A2
    sp:varName "source"^^xsd:string .
_:A3
    sp:varName "sink"^^xsd:string .
_:A4
    sp:varName "functionality"^^xsd:string .
_:A5
```

```
    sp:varName "sink"^^xsd:string .
_:A6
    sp:varName "lastEvent"^^xsd:string .
_:A7
    sp:varName "id"^^xsd:string .
_:A8
    sp:varName "tag"^^xsd:string .
_:A9
    sp:varName "datetime"^^xsd:string .
```

BIBLIOGRAPHY

[1] W3C Web Events Working Group Charter. URL http://www.w3.org/2010/webevents/charter/.

[2] OWL Web Ontology Language Use Cases and Requirements, 2004. URL http://www.w3.org/TR/webont-req/.

[3] Dean Allemang and Jim Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kauffman, 2nd edition, 2011. ISBN 978-0-12-385965-5.

[4] Timo Arnall. A graphic language for touch-based interactions. In *Proceedings of the Mobile Interaction with the Real World workshop (MIRW 2006)*, 2006.

[5] Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. iStuff: A Physical User Interface Toolkit for Ubiquitous Computing Environments. In *Proceedings of the ACM CHI 2003 Conference on Human Factors in Computing Systems*, pages 537–544, Ft. Lauderdale, Florida, USA, 2003.

[6] Rafael Ballagas, A. Szybalski, and A. Fox. Patch panel: enabling control-flow interoperability in ubicomp environments. *Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. Proceedings of the*, pages 241–252, 2004.

[7] Rafael Ballagas, Jan Borchers, Michael Rohs, and Jennifer G Sheridan. The smart phone : A ubiquitous input device. *IEEE Pervasive Computing*, 5(1):70–77, 2006.

[8] Jie Bao, Elisa F. Kendall, Deborah L. McGuinness, and Peter F. Patel-Schneider. OWL 2 Web Ontology Language: Quick Reference Guide, 2009. URL http://www.w3.org/TR/owl2-quick-reference/.

[9] Sara Bartolini, Bojan Milosevic, Alfredo D'Elia, Elisabetta Farella, Luca Benini, and Tullio Salmon Cinotti. Reconfigurable natural interaction in smart environments: approach and prototype implementation. *Personal and Ubiquitous Computing*, September 2011. ISSN 1617-4909.

[10] Len Bass, Ross Faneuf, Reed Little, Niels Mayer, Bob Pellegrino, Scott Reed, Robert Seacord, and Sylvia Sheppard. A metamodel for the runtime architecture of an interactive system. *SIGCHI Bulletin*, 24(1):32–37, 1992.

[11] Victoria Bellotti, Maribeth Back, W. Keith Edwards, Rebecca E. Grinter, Austin Henderson, and Cristina Lopes. Making sense of sensing systems: five questions for designers and researchers. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, number 1, pages 415–422, 2002.

[12] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[13] Sachin Bhardwaj, Tanir Özcelebi, Richard Verhoeven, and Johan Lukkien. Delay Performance in a Semantic Interoperability Architecture. *2011 IEEE/IPSJ International Symposium on Applications and the Internet*, pages 280–285, July 2011.

[14] Alan Blackwell and Thomas Green. A Cognitive Dimensions Questionnaire, 2007. URL https://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf.

[15] Alan F. Blackwell and Rob Hague. AutoHAN: an architecture for programming the home. *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 150–157, 2001.

[16] Bert Bongers and G. C van Der Veer. Towards a Multimodal Interaction Space: categorisation and applications. *Personal and Ubiquitous Computing*, 11(8):609–619, February 2007. ISSN 1617-4909.

[17] George E.P. Box and Norman R. Draper. *Empirical Model-Building and Response Surfaces*. 1987. ISBN 0-471-81033-9.

[18] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[19] M. E Bratman, D. J Israel, and M. E Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3):349–355, 1988.

[20] Bill Buxton. Natural User Interface: What's in a name?, 2010. URL http://www.designbyfire.nl/2010/program#bill.

[21] Muffy Calder, Phil Gray, and Chris Unsworth. Tightly coupled verification of pervasive systems. In *Proceedings of the Third International Workshop on Formal Methods for Interactive Systems (FMIS 2009)*, volume 22, 2009.

[22] Stuart K. Card, Jock D. Mackinlay, and George G. Robertson. A morphological analysis of the design space of input devices. *ACM Transactions on Information Systems (TOIS)*, 9(2):99, 1991. ISSN 1046-8188.

[23] Matthew Chalmers and Ian MacColl. Seamful and Seamless Design in Ubiquitous Computing. *Computer*, (Equator-03-005), 2003.

[24] Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. SOUPA: standard ontology for ubiquitous and pervasive applications. In *Proceedings of Mobile and Ubiquitous Systems: Networking and Services (MOBIQUITOUS 2004)*, pages 258–267, 2004.

[25] Steven Clarke. Measuring API Usability. *Dr Dobbs Journal*, 10(1), April 2004. ISSN 1361-4533.

[26] Joelle Coutaz, James L. Crowley, Simon Dobson, and David Garlan. Context is key. *Communications of the ACM*, 48(3):49–53, 2005.

[27] Mathieu D'Aquin and Aldo Gangemi. Is there beauty in ontologies ? *Applied Ontology*, 6(3/2011):165–175, 2011.

[28] Maurice M. de Ruiter. *Advances in Computer Graphics III*. Springer-Verlag, 1988. ISBN 0-387-18788-X.

[29] Anind K Dey. Context-aware computing. In John Krumm, editor, *Ubiquitous Computing Fundamentals*, pages 321–352. CRC Press, 2010. ISBN 978-1-4200-9360-5.

[30] Alan Dix. Upside down As and algorithms - computational formalisms and theory. In John Carroll, editor, *HCI Models, Theories and Frameworks: Toward a Multidisciplinary Science*, pages 381–429. Morgan Kauffman, San Francisco, USA, 2003.

[31] Alan Dix, Janet Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction*. Pearson Education Limited, 3rd edition, 2004.

[32] Alan Dix, Masitah Ghazali, Steve Gill, Joanna Hare, and Devina Ramduny-Ellis. Physigrams: modelling devices for natural interaction. *Formal Aspects of Computing*, 21(6): 613–641, December 2008. ISSN 0934-5043.

[33] Leigh Dodds and Ian Davis. *Linked Data Patterns*. 2011. URL `http://patterns.dataincubator.org`.

[34] Paul Dourish. *Where the Action Is*. MIT Press, September 2004. ISBN 0262541785, 9780262541787.

[35] Nick Drummond. The Manchester OWL Syntax, 2009. URL `http://www.co-ode.org/resources/reference/manchester_syntax/`.

[36] Emmanuel Dubois and Philip Gray. A design-oriented information-flow refinement of the ASUR interaction model. *Engineering Interactive Systems*, 4940/2008:465–482, 2008.

[37] W. Keith Edwards and Rebecca E Grinter. At home with ubiquitous computing: Seven challenges. *Ubicomp 2001: Ubiquitous Computing*, pages 256–272, 2001.

[38] Matti Etelapera, Jussi Kiljander, and Kari Keinanen. Feasibility Evaluation of M3 Smart Space Broker Implementations. *2011 IEEE/IPSJ International Symposium on Applications and the Internet*, pages 292–296, July 2011.

[39] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/-subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003. ISSN 03600300.

[40] Loe M.G. Feijs and R.C. Van Ommering. Relation partition algebra - mathematical aspects of uses and part-of relations. *Science of Computer Programming*, 33, 1999.

[41] Dieter Fensel. Triple-space computing: Semantic Web Services based on persistent publication of information. In *Intelligence in Communication Systems*, Lecture Notes in Computer Science, pages 43–53. Springer, 2004.

[42] James D. Foley, Victor L. Wallace, and Peggy Chan. The human factors of computer graphics interaction techniques. *IEEE Computer Graphics and Applications*, 4(11): 13–48, 1984.

[43] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics: principles and practice*. Addison-Wesley Publishing Company, Inc., 2nd editio edition, 1996. ISBN 0201848406.

[44] Joep J.W. Frens and Kees C.J. Overbeeke. Setting the Stage for the Design of Highly Interactive Systems. In *Proceedings of the International Association of Societies of Design Research (IASDR'09)*, pages 1–10, Seoul, Korea, 2009.

[45] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Inc., 1994. ISBN 0-201-63361-2.

[46] Aldo Gangemi and Peter Mika. Understanding the Semantic Web through Descriptions and Situations. In *Proceedings of ODBASE03 Conference*, pages 689—-706. Springer, 2003.

[47] Aldo Gangemi and Valentina Presutti. Content Ontology Design Patterns as Practical Building Blocks for Web Ontologies. In *Conceptual Modeling - ER 2008*, Lecture Notes in Computer Science, pages 128–141. Springer Berlin / Heidelberg, 2008.

[48] Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. The Belief-Desire-Intention Model of Agency. In *Intelligent Agents V: Agents, theories, architectures and languages*, Lecture Notes in Computer Science, pages 1–10. Springer, 1999.

[49] M.P. Georgeff and A.S. Rao. A profile of the Australian Artificial Intelligence Institute [World Impact]. *IEEE Expert*, 11(6):89, 1996. ISSN 0885-9000.

[50] J.J. Gibson. The Theory of Affordances. In Robert Shaw and John Bransford, editors, *Perceiving, Acting and Knowing: Toward an Ecological Psychology*, pages 67—-82. LEA, Hillsdale, 1977.

[51] T.R.G. Green and M. Petre. Usability Analysis of Visual Programming Environments : a 'cognitive dimensions' framework. *Applied Psychology*, (January):1–51, 1996.

[52] Adam Greenfield. *Everyware: The dawning age of ubiquitous computing*. New Riders, 2006.

[53] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. An Ontology-based Context Model in Intelligent Environments. In *Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 270—-275, 2004.

[54] John Hebeler, Matthew Fisher, Ryan Blace, and Andrew Perez-Lopez. *Semantic web programming*. Wiley Publishing, 2009. ISBN 047041801X.

[55] Martin Hepp. Ontologies: State of the Art, Business Potential, and Grand Challenges. In Martin Hepp, Pieter De Leenheer, Aldo De Moor, and York Sure, editors, *Ontology Management: Semantic Web, Semantic Web Services, and Business Applications*, pages 3–22. 2007. ISBN 9780387698991.

[56] Rinke Hoekstra. *Ontology Representation: design patterns and ontologies that make sense*. PhD thesis, University of Amsterdam, 2009.

[57] Rinke Hoekstra. Representing Social Reality in OWL 2. In *OWLED 2010, 7th International Workshop*, San Francisco, California, USA, 2010. URL `http://www.webont.org/owled/2010/papers/owled2010_submission_29.pdf`.

[58] Rinke Hoekstra and Joost Breuker. Polishing diamonds in OWL 2. *Knowledge Engineering: Practice and Patterns*, 5268/2008:64–73, 2008.

[59] Jukka Honkola, Hannu Laine, Ronald Brown, and Olli Tyrkko. Smart-M3 Information Sharing Platform. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 1041–1046, Riccione, Italy, 2010.

[60] Interaction Design Foundation. Interaction Design Quote. URL `http://www.interaction-design.org/quote/`.

[61] Thomas C. Jepsen. Just What Is an Ontology, Anyway? *IEEE IT Professional*, 11(5):22–27, 2009. ISSN 1520-9202.

[62] Michael Jeronimo and Jack Weast. *UPnP Design by Example*. Intel Press, 2009.

[63] B. Johanson and A. Fox. The Event Heap: a coordination infrastructure for interactive workspaces. *Proceedings Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 83–93, 2002.

[64] Matt Jones and Gary Marsden. *Mobile Interaction Design*. John Wiley and Sons, 2006.

[65] Satya Komatineni, Dave MacLean, and Sayed Hashimi. *Pro Android 3*. Apress, 2011. ISBN 1430232226.

[66] Mike Kuniavsky. *Smart Things: Ubiquitous Computing User Experience Design*. Morgan Kauffman, 2010. ISBN 978-0-12-374899-7.

[67] Matthijs Kwak, Gerrit Niezen, Bram J.J. van der Vlist, Jun Hu, and Loe M.G. Feijs. Tangible Interfaces to Digital Connections, Centralized versus Decentralized. In *Transactions on Edutainment V, Lecture Notes in Computer Science*. 2011.

[68] Craig Larman and Victor R. Basili. Iterative and Incremental Development: A brief history. *IEEE Computer*, 36 (6):47–56, 2003.

[69] Choonhwa Lee, Sumi Helal, and Wonjun Lee. Universal interaction with smart spaces. *IEEE Pervasive Computing*, 5(1):16–21, 2006.

[70] V Luukkala and I Niemelä. Enhancing a Smart Space with Answer Set Programming. *Semantic Web Rules*, LNCS 6403:89–103, 2010.

[71] Wendy E Mackay and Anne-Laure Fayard. HCI , Natural Science and Design : A Framework for Triangulation Across Disciplines. In *Proceedings of the 2nd conference on Designing interactive systems: processes, practices, methods, and techniques (DIS'97)*, pages 223–234, 1997.

[72] Jock Mackinlay, Stuart Card, and George Robertson. A Semantic Analysis of the Design Space of Input Devices. *Human-Computer Interaction*, 5(2):145–190, June 1990. ISSN 0737-0024.

[73] Irene Mavrommati, Achilles Kameas, and Panos Markopoulos. An editing tool that manages device associations in an in-home environment. *Personal and Ubiquitous Computing*, 8(3-4):255–263, June 2004. ISSN 1617-4909.

[74] David Merrill, Jeevan Kalanithi, and Pattie Maes. Siftables: towards sensor network user interfaces. In *Proceedings of the 1st international conference on Tangible and embedded interaction (TEI'07)*, pages 75–78, Baton Rouge, Louisiana, 2007. ACM. ISBN 978-1-59593-619-6.

[75] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the Fall Joint Computer Conference*, page 267. ACM Press, 1968.

[76] Mark W. Newman, Jana Z. Sedivy, Christine M. Neuwirth, W. Keith Edwards, Jason I. Hong, Shahram Izadi, Karen Marcelo, and Trevor F. Smith. Designing for serendipity: supporting end-user configuration of ubiquitous computing environments. *Proceedings of the conference on Designing interactive systems processes, practices, methods, and techniques - DIS '02*, pages 147—-156, 2002.

[77] Hung Q. Ngo, Anjum Shehzad, Kim Anh Pham, Maria Riaz, Saad Liaquat, and Sung Young Lee. Developing Context-Aware Ubiquitous Computing Systems with a Unified Middleware Framework. In *Proceedings of Embedded and Ubiquitous Computing (EUC 2004)*, pages 239–247, 2004.

[78] J Nielsen. A virtual protocol model for computer-human interaction. *International Journal of Man-Machine Studies*, 24(3):301–312, March 1986. ISSN 00207373.

[79] E. Niemela and T. Vaskivuo. Agile middleware of pervasive computing environments. In *Proceedings of the 2nd IEEE Anual Conference on Pervasive Computing and Communication Workshops (PERCOMW'04)*, pages 192–197, Orlando, FL, USA, March 2004.

[80] Gerrit Niezen, Bram J.J. van der Vlist, Jun Hu, and Loe M.G. Feijs. From events to goals: supporting semantic interaction in smart environments. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, pages 1029–1034, Riccione, Italy, 2010.

[81] Gerrit Niezen, Bram J.J. van der Vlist, Jun Hu, and Loe M.G. Feijs. Using semantic transformers to enable interoperability between media devices in a ubiquitous computing environment. In *Grid and Pervasive Computing*

Workshops, Lecture Notes in Computer Science, volume 7096, pages 44–53. Springer Berlin / Heidelberg, 2011.

[82] Gerrit Niezen, Bram J.J. van der Vlist, Sachin Bhardwaj, and Tanir Ozcelebi. Performance Evaluation of a Semantic Smart Space Deployment. In *4th International Workshop on Sensor Networks and Ambient Intelligence (SeNAmI 2012)*, Lugano, Switzerland, March 2012.

[83] Joshua Noble. *Programming Interactivity*. O'Reilly, 2009.

[84] Donald A. Norman. *The Design of Everyday Things*. MIT Press, 1998. ISBN 0-262-64037-6.

[85] Donald A. Norman. *The Invisible Computer*. MIT Press, 1999.

[86] N. Noy and A. Rector. Defining n-ary relations on the semantic web, 2006. URL `http://www.w3.org/TR/swbp-n-aryRelations`.

[87] Martin J O'Connor and Amar K Das. A Lightweight Model for Representing and Reasoning with Temporal Information in Biomedical Ontologies. In *International Conference on Health Informatics (HEALTHINF)*, Valencia, Spain, 2010.

[88] Ian Oliver and Jukka Honkola. Personal semantic web through a space based computing environment. In *Middleware for the Semantic Web, Second IEEE International Conference on Semantic Computing*, Santa Clara, CA, USA, August 2008. URL `http://arxiv.org/pdf/0808.1455`.

[89] Dan R. Olsen, S. Travis Nielsen, and David Parslow. Join and capture: a model for nomadic interaction. In *Proceedings of the The 14th Annual ACM Symposium on User Interface Software and Technology (UIST'01)*, volume 3, pages 131–140, 2001.

[90] Tim O'Reilly and John Battelle. Web Squared: Web 2.0 Five Years On. Technical report, O'Reilly, 2009.

[91] Jeffrey S. Pierce and Heather Mahaney. Opportunistic Annexing for Handheld Devices : Opportunities and Challenges. Technical report, Proceedings of HCIC, 2003.

[92] Erika Shehan Poole, Marshini Chetty, Rebecca E Grinter, and W Keith Edwards. More Than Meets the Eye : Transforming the User Experience of Home Network Management. In *Proceedings of the 7th ACM Conference on Designing Interactive Systems (DIS'08)*, pages 455–464, Cape Town, South Africa, 2008. ACM.

[93] Davy Preuveneers and Yolande Berbers. Encoding semantic awareness in resource-constrained devices. *IEEE Intelligent Systems*, 23(2):26–33, 2008.

[94] Anand Ranganathan, Jalal Al-Muhtadi, and Roy H. Campbell. Reasoning about Uncertain Contexts in Pervasive Computing Environments. *IEEE Pervasive Computing*, 3(2):62–70, 2004. ISSN 1536-1268.

[95] Stefan Rapp. Spotlight Navigation : a pioneering user interface for mobile projection. In *Proceedings of Ubiprojection*, Helsinki, Finland, 2010.

[96] Jun Rekimoto, Yuji Ayatsuka, Michimune Kohno, and Hauro Oba. Proximal interactions: A direct manipulation technique for wireless networking. In *Proceedings of INTERACT2003*, pages 511—-518, 2003.

[97] Charles Rich. Building Task-Based User Interfaces with ANSI/CEA-2018. *IEEE Computer*, 42(8):20–27, 2009. ISSN 0018-9162.

[98] Martin P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34, November 2009. ISSN 0740-7459.

[99] Ben Rubin. 5 Steps to Phasing Sleep, 2009. URL `http://blog.myzeo.com/5-steps-to-phasing-sleep/`.

[100] Marta Sabou. Smart objects : Challenges for Semantic Web research. *Semantic Web*, 1(1-2):127–130, 2010.

[101] Ansgar Scherp, Carsten Saathoff, Thomas Franz, and Steffen Staab. Designing core ontologies. *Applied Ontology*, 6: 177–221, 2011.

[102] Toby Segaran and Jeff Hammerbacher. Beautiful Data: The Stories Behind Elegant Data Solutions. O'Reilly, 2009.

[103] Toby Segaran, Colin Evans, and Jamie Taylor. *Programming the Semantic Web*. O'Reilly, 2009. ISBN 978-0-596-15381-6.

[104] Juan Sequeda. SPARQL 101, 2012. URL `http://www.cambridgesemantics.com/semantic-university/sparql-101`.

[105] Orit Shaer, Nancy Leland, Eduardo H. Calvillo-Gamez, and RobertJ.K. Jacob. The TAC paradigm: specifying tangible user interfaces. *Personal and Ubiquitous Computing*, 8 (5):359–369, July 2004. ISSN 1617-4909.

[106] CE Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423, 623–423, 1948.

[107] Ryan Shaw, Raphael Troncy, and Lynda Hardman. LODE : Linking Open Descriptions of Events. In *The Semantic Web*, Lecture Notes in Computer Science, pages 153–167. Springer Berlin / Heidelberg, 2009.

[108] Sarah Slobin. The 7 1/2 steps to successful infographics, 2010. URL `http://visitmix.com/Articles/seven-and-a-half-steps-to-successful-infographics`.

[109] Harold Thimbleby. *Press on: Principles of interaction programming*. MIT Press, 2007. ISBN 978-0-262-20170-4.

[110] Manas Tungare, Pardha S Pyla, Miten Sampat, and Manuel A Perez-Quinones. Syncables : A Framework to Support Seamless Data Migration Across Multiple Platforms. In *IEEE International Conference on Portable Information Devices (IEEE Portable)*, 2007.

[111] Ubiquity Staff. The new computing. *Ubiquity*, September 2002.

[112] B. Ullmer and H. Ishii. Emerging frameworks for tangible user interfaces. *IBM Systems Journal*, 39(3):915, 2000. ISSN 0018-8670.

[113] UPnP Forum. UPnP Remote UI Client and Server V 1.0, . URL `http://www.upnp.org/specs/rui/remoteui/`.

[114] UPnP Forum. UPnP Standards: Device Control Protocols, . URL `http://upnp.org/sdcps-and-certification/standards/sdcps/`.

[115] G. C van Der Veer and Mari Carmen Puerta Melguizo. Mental Models. In *The Human-Computer Inteaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications*, pages 52–80. 2003.

[116] Bram J.J. van der Vlist. *Designing Semantic Connections: Explorations, Theory and a Framework for Design*. PhD thesis, Eindhoven University of Technology, 2012.

[117] Bram J.J. van der Vlist, Gerrit Niezen, Jun Hu, and Loe M.G. Feijs. Semantic Connections: Exploring and Manipulating Connections in Smart Spaces. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, pages 1–4, Riccione, Italy, June 2010.

[118] Bram J.J. van der Vlist, Gerrit Niezen, Jun Hu, and Loe M.G. Feijs. Design Semantics of Connections in a Smart Home Environment. In *Design and Semantics of Form and Movement (DeSForM2010)*, pages 48–56, Lucerne, Switzerland, 2010.

[119] Bram J.J. van der Vlist, Gerrit Niezen, Jun Hu, and Loe M.G. Feijs. Interaction Primitives: Describing Interaction Capabilities of Smart Objects in Ubiquitous Computing Environments. In *IEEE AFRICON 2011*, Livingstone, Zambia, 2011.

[120] Bram J.J. van der Vlist, Gerrit Niezen, Stefan Rapp, Jun Hu, and Loe M.G. Feijs. Controlling Smart Home Environments with Semantic Connections: a Tangible and an AR Approach. In *7th International Workshop on the Design & Semantics of Form & Movement (DeSForM)*, Wellington, New Zealand, April 2012.

[121] Claudia Villalonga, Martin Strohbach, Niels Snoeck, M. Sutterer, M. Belaunde, E. Kovacs, A. Zhdanova, L. Goix, and O. Droegehorn. Mobile ontology: Towards a standardized semantic model for the mobile domain. In *Service-Oriented Computing-ICSOC 2007 Workshops*, pages 248–257. Springer, 2009.

[122] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, 1991.

[123] Stephan A.G. Wensveen. *A Tangibility Approach to Affective Interaction*. PhD thesis, Delft University of Technology, 2005.

[124] Stephan A.G. Wensveen, J P Djajadiningrat, and Kees C.J. Overbeeke. Interaction Frogger : A Design Framework to Couple Action and Function through Feedback and Feedforward. In *Proceedings of the 5th Conference on Designing Interactive Systems (DIS'04)*, pages 177–184, Cambridge, MA, 2004. ACM. ISBN 1581137877.

[125] Terry Winograd. *Bringing design to software*. Addison-Wesley Publishing Company, Inc., 1996.

[126] Terry Winograd, Daniel M. Russell, and Norbert A. Streitz. Building disappearing computers. *Communications of the ACM*, 48(3):42–48, 2005.

[127] Alex Woodie. Jeff Jonas Explores the Nature of Data in COMMON Keynote, 2009. URL `http://www.itjungle.com/tfh/tfh051809-story03.html`.

[128] Juan Ye, Simon Dobson, Lorcan Coyle, and Paddy Nixon. Ontology-Based Models in Pervasive Computing Systems. *The Knowledge Engineering Review*, 22(04):315–347, 2007.

[129] Y. Yesha, F. Perich, T. Finin, and A. Joshi. On data management in pervasive computing environments. *IEEE Transactions on Knowledge and Data Engineering*, 16(5):621–634, 2004. ISSN 1041-4347.

[130] Gottfried Zimmermann and Gregg Vanderheiden. The Universal Control Hub : An Open Platform for Remote User Interfaces in the Digital Home. In *Proceedings of the 12th international conference on Human-computer interaction: interaction platforms and techniques (HCI'07)*, pages 1040–1049, 2007.

ACKNOWLEDGMENTS

The work presented in this thesis would not have been possible without the help of my supervisors, co-workers, colleagues, and friends.

First, I would like to thank my first supervisor Prof. Loe Feijs, my second supervisor Prof. Panos Markopoulos, and my co-supervisor Dr. Jun Hu. I would also like to thank the members of my reading committee: Prof. Harold Thimbleby, Dr. Stefan Wensveen and Prof. Johan Lukkien. Thank you for all your help, suggestions and guidance.

Bram van der Vlist — who was my teammate on this adventure — without you this would definitely not have been possible.

I would like to thank Aly Syed, Riccardo Trevisan, Sriram Srinivasan, Hans van Amstel, Stefan Rapp, Sachin Bhardwaj and Tanir Ozcelebi for their valuable contributions to the smart home pilot.

While working on the software tools and ontologies described in this thesis, Holger Knublauch and Scott Henninger from TopQuadrant were always ready to provide me with assistance.

I will always remember my PhD colleagues who I shared an office with at the university: Alex Juarez, Sibrecht Bouwstra, Roman Gorbunov, Jan Gillesen, Misha Croes and Marija Nakevska. You guys made my three years at TU Eindhoven very enjoyable and a lot of fun. Also a sincere thank you to Ellen Konijnenberg for helping me with all the organisational issues.

I would like to thank my parents and my brother: *Ma, Pa en Riaan - baie dankie vir julle ondersteuning and dat julle in my glo.* This thesis is dedicated to my fiancée — Margriet van de Goor, *ik hou van je*!

CURRICULUM VITAE

Gerrit Niezen was born in Pretoria, South Africa, on 29 September 1982. He received BEng Computer Engineering and MEng Computer Engineering (with distinction) degrees from the University of Pretoria in South Africa. After spending a brief period in industry, he returned to the University of Pretoria as a lecturer, teaching undergraduate classes in software engineering and network security, as well as a postgraduate class in wireless sensor networks. Gerrit's research interests include interaction design, semantic web technologies, wireless sensor networks, ubiquitous computing and gesture recognition.

He joined the Designed Intelligence research group at TU Eindhoven in July 2009. His research involved determining how user interaction in a smart space can be better supported by ontology modelling. His research is part of a larger EU project to improve interoperability between devices in a smart environment. This technological design is the results from the work in the years 2009 to 2012.