

Diss. ETH No. 22203

Interacting with the Web of Things

A dissertation submitted to
ETH ZURICH

for the degree of
DOCTOR OF SCIENCES

Presented by
Simon Mayer
MSc in Computer Science, ETH Zurich
born July 16, 1987
citizen of Austria

accepted on the recommendation of
Prof. Dr. Friedemann Mattern, examiner, ETH Zurich
Prof. Dr. Sanjay Sarma, co-examiner, Massachusetts Institute of Technology
Prof. Dr. Gustavo Alonso, co-examiner, ETH Zurich

2014

A fundamental paradigm shift is currently taking place in the field of computing: due to the miniaturization of computing devices and the proliferation of embedded systems, tiny, networked computers can now be easily integrated into everyday objects, turning them into *smart things*. In the resulting *Internet of Things*, physical items are no longer disconnected from the virtual world but rather become accessible through computers and other networked devices, and can even make use of protocols that are widely deployed in the World Wide Web, in a paradigm that we call the *Web of Things*. Eventually, smart things will be able to communicate, analyze, decide, and act – and thereby provide an invisible background assistance that should make life more enjoyable, entertaining, and also safer. However, in an environment that is populated by hundreds of Web-enabled smart things, it will become increasingly difficult for humans to interact with devices that are relevant to their current needs, and to find, select, and control them.

The objective of this thesis is to investigate how human users could be enabled to conveniently interact with individual smart objects in their surroundings and to interconnect devices and configure the resulting *physical mashups* to perform higher-level tasks on their behalf. To achieve basic interoperability between devices, we rely on the World Wide Web with its proven protocols and architectural patterns which emphasize scalability, generic interfaces, and loose coupling between components.

As a first step to facilitate the interaction with smart things on top of the basic Web principles, we propose the embedding of metadata for automatically generating user interfaces for smart devices. Our specific approach enables not only the generation of more intuitive graphical widgets but also the mapping of interactive components to gesture-based, speech-based, and physical interfaces by describing the *high-level interaction semantics* of smart devices instead of specifying purely interface-specific information. The provisioning of an interaction mechanism with a smart object is thus reduced to the embedding of simple interaction information into the representation of the smart thing. Before users can start *interacting* with a smart device, it must, however, first be *selected*. To permit users to choose which of the many smart objects in their surroundings should be involved in an interaction, we propose to use technologies for *optical image recognition*.

The visual selection of smart things and automatically generated user interfaces enable end users to conveniently interact with individual services in their surroundings that are embodied as specific physical objects. To complement the direct interaction with smart devices, the second part of this thesis focuses on more complex use cases where

multiple smart objects must collaborate to achieve the user's goal. Such situations arise, for instance, in home or office automation scenarios, or in smart factories, where machines or assembly lines could adjust to better support the operator.

To put users more in control of entire environments of smart devices, we present a system that records *interactions between smart things and with remote services* and displays this data to users in real time. To do this, we use an augmented reality overlay on the camera feed of handheld or wearable devices such as smartphones and smartglasses. Next, we propose a management infrastructure for smart things that makes the services they offer discoverable and composable, and fully integrates them with more traditional Web-based information providers. This system enables humans to *find and use* data and functionality provided by physical devices and allows machines to support users in finding services within densely populated smart environments and even to discover and use required services themselves, on behalf of the user. The basis for these applications is a generic mechanism that allows smart devices to provide semantic descriptions of the services they offer. Specifically, our infrastructure supports the embedding of *functional semantic metadata* into smart things that describes which functionality a concrete object provides and how to invoke it. Based on this metadata, a semantic reasoning component can find out which composite tasks can be achieved by a user's smart environment and can provide instructions about how to reach concrete goals, thus enabling the *configuration of entire smart environments* for end users.

As a concrete use case, we present a platform that applies our proposed interaction modes with smart things to automobiles: a mobile application recognizes cars, downloads information about them from a back-end server, and displays this information – as well as interaction capabilities with the car and its services – on the user's interface device. The back-end server furthermore exposes functional metadata about the capabilities of individual cars to make their services automatically usable within physical mashups. Finally, it records client interactions to enable car owners to monitor in real time who accesses which kind of data and services on their vehicles.

The overarching objective of this thesis is to show how current technologies could support the interaction of end users with Web-enabled smart devices. To achieve this, we make use of a number of technologies from different areas of the computer science discipline: A management infrastructure makes smart things discoverable for human users and machines and builds upon current research in the distributed systems domain. State-of-the-art computer vision technologies allow users to select devices in their environment using handheld or wearable computers such as smartphones or smartglasses. Novel methods from the field of computer-human-interaction enable the embedding of metadata that allows for automatically generating user interfaces. Finally, semantic technologies enable flexible compositions of smart things that collaborate to achieve the user's goal.

Anhaltende Fortschritte bei der Miniaturisierung von Mikroelektronik und Sensorik sowie bei Kommunikationstechnologien ermöglichen die Einbettung von vernetzten Kleinstcomputern in Alltagsgegenstände. Solche sogenannten „Smart Things“ – schlaue, wenn auch nicht im eigentlichen Sinne intelligente Dinge – sind Geräte, die mit einer virtuellen Präsenz im *Internet der Dinge* gepaart sind. Sie können miteinander und mit Menschen kommunizieren, ihre Umwelt durch Sensoren wahrnehmen, autonom Entscheidungen treffen und auf die Welt mittels ihrer Aktoren einwirken. Wenn dies im Sinne der menschlichen Benutzer geschieht, agieren vernetzte, schlaue Dinge wie eine unsichtbare Hintergrundassistentin, die unser Leben angenehmer, unterhaltsamer und auch sicherer machen kann. Sollten diese Geräte zudem die Fähigkeit mitbringen, Kommunikationsprotokolle, welche im World Wide Web eingesetzt werden, zu verwenden, so sprechen wir vom „Web of Things“, dem Web der Dinge.

Das Ziel dieser Arbeit ist, zu untersuchen, wie menschliche Benutzer dabei unterstützt werden können, sich in Umgebungen zurechtzufinden, die hunderte schlauer Dinge enthalten. In solchen *smarten Umgebungen* ist es für Benutzer insbesondere schwierig, jene Geräte, die sie gerade benötigen, effizient aufzufinden und intuitiv auszuwählen, und mit ihnen in geeigneter Weise zu interagieren. Zudem soll es Benutzern ermöglicht werden, Geräte in derartigen Umgebungen so zu konfigurieren, dass sie in kooperativer Weise Aufgaben erledigen können, welche für ein einzelnes schlaues Ding zu komplex sind. Die Basis für diese Arbeit bildet dabei das World Wide Web, das durch seinen Aufbau und seine offenen Protokolle eine grundlegende Interoperabilität zwischen schlauen Dingen ermöglicht.

Zunächst wird in der vorliegenden Arbeit die direkte, unmittelbare Interaktion von Benutzern mit schlauen Dingen behandelt: Hierfür stellen letztere Metadaten zur Verfügung, welche die *automatische Erzeugung von Nutzungsschnittstellen* auf tragbaren Geräten wie Smartphones, Tablets und Smartglasses ermöglichen. Indem schlaue Dinge ihre Interaktionssemantik auf hoher Ebene beschreiben, anstatt nur schnittstellenspezifische Informationen bereitzustellen, ermöglicht unser Konzept nicht nur die automatische Erzeugung von grafischen Widgets, sondern gleichzeitig auch gestenbasierte Interaktion sowie Sprachsteuerung. Darüber hinaus vereinfacht unser Ansatz das Beschreiben der Interaktionssemantik selbst, sodass dies sogar Laien ermöglicht wird. Bevor allerdings eine Nutzungsschnittstelle geladen werden kann, um mit einem schlauen Gegenstand zu interagieren, muss dieser vom Benutzer ausgewählt werden. Hierfür verwenden wir aktuelle

Technologien aus der optischen Bilderkennung: Um die *Interaktion mit einem Gerät zu initiieren*, müssen Benutzer in unserem Ansatz lediglich mit der Kamera ihres Smartphones oder Tablets auf das Gerät zielen – falls sie über Smartglasses verfügen, ist es ausreichend, das Gerät einfach nur anzusehen.

Im zweiten Teil der Arbeit wird die Interaktion mit smarten Umgebungen als Ganzes untersucht, wobei mehrere schlaue Dinge selbstständig zusammenarbeiten sollen, um Benutzer bei komplexeren Aufgaben zu unterstützen. Mithilfe der in dieser Arbeit entwickelten Technologien können beispielsweise Automatisierungsszenarien zu Hause und in Fabrikumgebungen umgesetzt werden – dort sollen sich in Zukunft einzelne Geräte oder ganze Fertigungsanlagen automatisch abstimmen und schnell anpassen, um den Produktionsprozess effizienter zu gestalten, insbesondere bei kleinen Losgrößen. Zunächst beschreiben wir eine Managementinfrastruktur, die das Auffinden und Zusammensetzen von Diensten, die von schlaue Dingen bereitgestellt werden, vereinfacht. Dieses System unterstützt insbesondere die *Suche nach Geräten und Diensten* in dichten smarten Umgebungen und ermöglicht durch *eingebettete semantische Beschreibungen* anderen Geräten, diese im Sinne des Benutzers anzusteuern. Auf Basis dieser semantischen Metadaten, welche die Funktionalität von einzelnen schlaue Geräten charakterisieren, stellen wir sodann ein System vor, welches ermitteln kann, welche Aufgaben Geräte in schlaue Umgebungen *gemeinsam* erledigen können. Hierfür gibt der Benutzer lediglich den erwünschten Zielzustand seiner Umgebung an, und unser System findet mithilfe eines semantischen Reasoners selbst heraus, ob und wie dieser Zustand erreicht werden kann. Um die Kontrolle über solche dynamischen Abläufe in smarten Umgebungen zu behalten, stellen wir ein System vor, welches Interaktionen zwischen schlaue Dingen protokolliert und sie in Echtzeit auf Geräten wie Smartphones oder Smartglasses in Form einer Augmented-Reality-Einblendung visualisiert.

Abschließend wird ein konkreter Anwendungsfall der in der vorliegenden Arbeit vorgestellten Konzepte und Technologien behandelt: „Connected Cars“. Eine von uns entwickelte Anwendung erkennt Fahrzeuge optisch und erzeugt automatisch Schnittstellen, welche benutzerfreundliche Interaktion mit Fahrzeugsensoren und -aktoren ermöglichen – beispielsweise das Auslesen des aktuellen Tankfüllstands und des Treibstoffverbrauchs oder die Bedienung der Fahrzeugverriegelung. Zusätzlich visualisiert die Anwendung Interaktionen zwischen Fahrzeugen und anderen schlaue Dingen, wie auch mit Online-Diensten, um insbesondere unberechtigte Zugriffe auf die durch das Fahrzeug bereitgestellten Daten und Dienste aufzuzeigen. Des Weiteren ermöglichen eingebettete semantische Metadaten die Einbindung von Fahrzeugdaten in zusammengesetzten Anwendungen, welche komplexe Aufgaben übernehmen können.

Das übergeordnete Ziel dieser Arbeit besteht darin, zu zeigen, wie aktuelle Technologien aus verschiedenen Bereichen der Informatik die Interaktion zwischen Benutzern und schlaue Dingen substantiell vereinfachen können. Die Arbeit vereint dazu Methoden aus der grafischen Datenverarbeitung, künstlichen Intelligenz, Mensch-Maschine-Interaktion und verteilten Systemen, um Nutzern das Auffinden und Auswählen von schlaue Dingen zu ermöglichen und mit ihnen, sowie mit smarten Umgebungen insgesamt, zweckmäßig und effizient zu interagieren.

Acknowledgements

I greatly enjoyed the last four years that I spent working on this thesis thanks to the company and support of colleagues, professors, and loved ones.

First and foremost, I would like to express my very great appreciation to my supervisor Prof. Friedemann Mattern: Thank you for giving me the freedom to work on topics that I deem important and intriguing, for your support whenever I asked for it, and for all the fun during our discussions of items unrelated to computer science. I cannot imagine a better adviser for a PhD student.

I would like to thank Prof. Sanjay Sarma for being on my thesis committee and for welcoming me to his lab at Massachusetts Institute of Technology two years ago for a fascinating time: I very much enjoyed experiencing the dynamics of the Field Intelligence Lab that did not stop short of our walking meetings. I am also thankful to Prof. Gustavo Alonso: even before you joined my thesis committee, I enjoyed our collaboration on the department's panels.

Much of the work that I present in this thesis has been made possible only through international collaborations. I would like to express my gratitude to the CloudThink team: Erik Wilhelm at Singapore University of Technology and Design and Josh Siegel at MIT. I greatly appreciate the advice and support given by Anind K. Dey from Carnegie-Mellon University when jointly working on ways to create better smart things user interfaces for humans, and beyond. I am particularly grateful for the chance to collaborate with Ruben Verborgh from Ghent University on putting humans in control of their smart environments.

I want to thank my friends and colleagues in the Distributed Systems Group at ETH: Matthias Kovatsch, for the joyful moments we shared over the years here in Switzerland and abroad. Gábor Sörös, for enjoying life together during and when not jointly working on computer vision systems. Wilhelm Kleiminger, for making our way together whilst at times even agreeing on how we should go about that. Christian Beckel, for our conversations on and off the slopes. I thank my officemates Alexander Bernauer and Leyna Sadamori for the pleasant and stimulating work atmosphere we created together. And Dominique Guinard and Vlad Trifa, for introducing me to the fascinating field of the Web of Things and our time together in the WoT community. I further like to express my thanks to my other friends and colleagues in the DSG: Robert Adelman, Philipp Bolliger, Hông-Ân Cao, Christian Floerkemeier, Marian George, Steve Hinske, Anwar Hithnawi, Iulia Ion, Marc Langheinrich, Benedikt Ostermaier, Christof Roduner, Kay Römer, Silvia Santini, Hossein Shafagh, and Markus Weiss.

I had the pleasure of supervising a large number of student projects during my time at ETH and want to express my gratitude to all the students who contributed to several parts of this thesis: Andreas Tschofen, Yassin Nasir Hassan, Nadine Inhelder, Gianin Basler, Markus Schalch, Adrian Wicki, Michael Och, Simon Jutz, Bram Scheidegger, Claude Barthels, Sezer Güler, Raffael Buff, and Marco Poltera. Each of you has taught me a bit about myself as well. My special thanks go to David Karam and Markus Sutter.

I would like to thank the staff of ETH Zurich for providing one of the best scientific working environments worldwide, and in particular Barbara von Allmen Wilson and Denise Spicher. I also want to thank Florian Michahelles for giving me the motivational jolt that carried me through the last phase of working on this thesis. I am thankful to the Swiss National Science Foundation for supporting my work financially and without any hassle over the last four years.

Finally, I thank my family and friends for your support, your sympathy, and understanding. Anna-Lena, I cannot imagine a better partner than you.

1	Introduction and Motivation	1
1.1	Current Developments and Challenges	2
1.2	Research Goals and Contributions	4
1.3	Thesis Outline	5
2	Connecting Smart Things to the Web	7
2.1	Convergence in the Internet of Things	7
2.2	The Web of Things	8
2.3	Representational State Transfer	9
2.4	Summary	15
3	User Interfaces for Smart Things	17
3.1	Terminology	19
3.2	Interaction Semantics and Atomic Interactive Components	20
3.3	Describing Interaction Semantics	22
3.4	Elements of a Semantic Interface Description Language	27
3.5	A Generic Mobile User Interface for Smart Things	29
3.6	Evaluation	32
3.7	Related Work	39
3.8	Summary	41
4	Object Recognition for Direct Interaction with Smart Things	43
4.1	Interacting with Smart Environments	43
4.2	Device Selection using Visual Object Recognition	45
4.3	Foundations of Visual Object Recognition	46
4.4	Comparison of Feature Detectors and Descriptors	51
4.5	A Universal Remote Control for Smart Things	57
4.6	Summary	63
5	Real-time Visualization of Device Interactions	65
5.1	Eliciting Device Interactions in Smart Environments	65

5.2	Collecting Network Traffic Data	69
5.3	Visualizing Interactions	73
5.4	Applications	81
5.5	Summary	83
6	A Web-based Infrastructure for the Internet of Things	85
6.1	Middlewares for the Internet of Things	87
6.2	Finding and Describing Smart Devices	88
6.3	A Web-based Infrastructure for Smart Things	89
6.4	Deployment and Evaluation	106
6.5	Summary	110
7	Service Composition in the Web of Things	113
7.1	Service Composition for Smart Things	114
7.2	A Computational Marketplace for REST Services	124
7.3	Semantics-based Service Composition	134
7.4	Making Semantic Technologies Usable for End Users	149
7.5	Summary	154
8	Case Study: Interacting with Smart Cars	155
8.1	The CloudThink Platform	156
8.2	CloudThink Applications	159
8.3	Summary	165
9	Conclusions and Outlook	167
9.1	Interacting with Individual Smart Things	167
9.2	Configuring and Managing Smart Environments	168
9.3	Future Work	169
	Bibliography	173
	Appendices	199
A	Interaction Abstraction Schemas	199
B	Evaluation Results: Visual Object Recognition	217
C	Free and Open-source Software	221

CHAPTER 1

Introduction and Motivation

Augmenting physical objects by embedding communication and information technology and thus transforming them into *smart things* enhances their utility beyond their traditional use and generates substantial added value for individuals as well as for enterprises. Such smart things are capable of perceiving their context using sensors, interacting with their surroundings, making autonomous decisions, and of communicating with each other and with humans. They are expected to form the basis of a responsive and adaptable computing infrastructure that is woven into the fabric of everyday life. On the long run, it is expected that this development will lead to the formation of a world-wide distributed system of smart objects that is several orders of magnitude larger than the Internet [127], and will have a huge economic impact: Gartner, a research firm, predicts that by the year 2020, the total economic value-added that can be attributed to the proliferation of smart devices will be \$1.9 trillion dollars across a broad range of industries including healthcare, retail, and transportation [275].

Isolated smart devices can already provide useful services to human users. However, the real potential of embedding smart things in our everyday environments lies in the *interconnection* of the services they provide: physical objects will no longer be disconnected from the virtual world but rather become accessible through computers and other networked devices [126]. Enabling two-way communication between people and smart things or even interactions between objects that do not require human administrators will allow for sophisticated applications across virtual/physical boundaries. We expect that one of the first effects of this trend will be a transformation of our private homes [29] and that it will have a profound impact on industrial environments as well, for instance by supporting the rapid reconfiguration of manufacturing systems [111]. Smart devices that are aware of their surroundings will also be valuable in the healthcare domain, both in the context of ambient assisted living and by supporting doctors in medical environments. At home and at our workplaces, we expect that the proliferation of smart objects will significantly affect our daily lives as we will be able to use ubiquitous computing devices for interacting with the real world from almost anywhere, at any time. For instance, in smart homes, connected household appliances and entertainment devices will allow for

services such as remote monitoring and control, social media integration, and over-the-air firmware updates – potentially, the entire control logic of an appliance could be hosted remotely, thereby making local updates redundant altogether [107]. Smart electricity meters will enable us to query our environment for its current electricity consumption [80], have the system propose ideas for saving energy, and immediately implement our decisions by configuring devices such as smart thermostats. Information about the occupancy state of a smart home can be used to automatically infer heating schedules that reduce energy waste while still providing a comfortable indoor temperature level [102]. Meanwhile, in the background, a *smart grid* will optimize the distribution of electrical power and will allow for applications such as peak leveling and time-of-use pricing – for instance, to use energy more efficiently, cold appliances can be remotely controlled by utility companies to adapt their cooling cycles and avoid peak loads in the energy grid [10]. It will probably also become possible to query search engines for the location and state of many physical things [60, 126, 136, 167], thereby supporting users in finding and interacting with smart devices. Eventually, we expect that some things will be aware of relevant aspects of their context and thereby able to decide and act by themselves¹ – this emerging invisible background assistance will enable a plethora of applications beyond the smart home domain and ranging from improving energy efficiency to entertainment, security, and smart manufacturing.

1.1 Current Developments and Challenges

Some effects of the increasing interconnection of everyday devices are already apparent today: in the last few years, more and more everyday devices such as home appliances, consumer electronics equipment, and cars as well as industrial machines are being connected to the Internet [126], and the World Wide Web [78]. Having become a commodity in the developed world, Internet access is now spreading to embedded devices that can support the IP protocol despite their often constrained resources [240]. This development, termed the *Internet of Things (IoT)* [66, 126], represents the next step in the evolution of the Internet by enabling the physical world to produce and consume data automatically and communicate information to anyone and anything. The long-term implications of the development of Internet-enabled smart environments are manifold: Companies, for instance, will benefit from the ability to react to events in the physical world that brings along the possibility to control and manage the underlying processes in an informed and rapid or even automatic manner. For individuals, the consequences range from the empowerment of consumers to new ways of organizing society as a whole within a process that represents an important technical and social challenge [126]. The IoT is a “means of enriching the Internet with trillions of new nerve endings” [58], a metaphor that has also been taken up by large companies such as IBM within its *Smarter Planet* initiative [282].

¹Within the context of the “Social Web of Things” project, researchers at the Ericsson User Experience Lab have produced a video that represents a compelling aggregation of ideas at the core of this development [273]. I personally value this video a lot, both as a way of introducing many of the core ideas of ubiquitous computing to novices, and as a reminder of what the final outcome of work in this research domain could look like.

Linked together, these nerve endings can provide humans with a tool that opens the door to many new findings, applications, benefits, and also risks [165]. Especially as smart devices may handle context information (e.g., individuals' whereabouts) or may be closely interlinked with critical infrastructures (e.g., power supply) [266], this development gives rise to severe security and privacy issues [109].

In this thesis, we focus on another important challenge in the ubiquitous computing domain that concerns the *interaction* of human users with smart environments. Particularly in the smart home domain, some experts already issue warnings about a possible loss of control by smart home inhabitants [185] due to the increased difficulty of managing and interacting with smart devices. To make the IoT widely usable and drive its successful adoption in people's homes and at their workplaces, the human element must be taken into account: it is crucial that the effort required from end users to set up smart devices, interact with them, and configure them to behave in smart ways remains manageable.

A first step to achieve this is to consider the direct interaction of humans with smart devices: current studies in the field of Human-Computer Interaction (HCI) [108] provide valuable insights regarding how to enable users to interact with embedded and thus virtually *disappearing computers*. This term refers to the miniaturization of processing units and their integration into everyday objects as well as to computers that are not anymore perceived as such (mental disappearance) [218]. Especially in the first case, current developments thus lead to user interfaces that are much less palpable and obvious than those of traditional computing devices – in particular when considering environments that are populated by many heterogeneous smart things, it is difficult for users to find and utilize services that provide the functionality they require [219]. This necessitates steps to enable the intuitive interaction with smart objects when digitally augmenting them [94]. In this context, both the *selection* of smart devices to interact with and the provisioning of *suitable and intuitive user interfaces* must be considered.

On the longer term, we believe that this requirement does not only apply to the direct interaction of humans with individual smart things but also to interactions with entire smart environments. Especially the *coordination of device collaborations* is challenging for end users [29] and we therefore believe that it would be beneficial to push the management of such compositions to a more abstract level by allowing users to specify goals with respect to their environment instead of having them combine devices and services themselves in a process-driven way. Possible applications for this approach include individual well-being such as controlling the ambient temperature, and reconfigurations of smart environments, for instance in industrial automation scenarios. In all these applications, making smart things collaborate to achieve higher-level goals must be simple enough for end users.

In summary, it is not yet clear how the interaction with smart devices can be facilitated for human users nor how networked devices will be enabled to interact in smart ways with each other on behalf of the user – providing answers to these questions is, however, highly relevant to ensure that users want to adopt the IoT in their homes and workplaces, and thus to unlock the potential benefits of a highly connected smart world.

1.2 Research Goals and Contributions

The overall goal of this thesis is to investigate how human users can be supported in their interaction with individual smart devices and with entire smart environments. Specifically, the research questions that we consider relate to the *selection* of smart things, the proposition of *suitable user interfaces* for them, the *administration* of devices within densely populated smart environments, and the *collaboration* of multiple smart things to reach higher-level goals.

Selecting Smart Things: *How can we enable everyday users to initiate interactions with individual smart devices?*

We address the question of how end users can be enabled to intuitively select devices in their smart environment that provide a desired functionality. To achieve this, we propose to use methods from the field of visual object recognition. By deploying these on users' handheld or wearable devices, we enable them to select smart things by pointing the camera of the device at an object or – in the case of smartglasses – by merely looking at a smart device to interact with.

User Interfaces for Smart Things: *How can smart things provide information about adequate interfaces to interact with them?*

After having selected a smart thing or service to interact with, human users will in most cases require a user interface to control that device. We propose that smart things embed information about suitable interfaces that can be looked up and rendered on demand by mediating devices such as smartphones or smartglasses. To this end, we propose an interaction description language for smart things that is simple to create and embed into devices yet expressive enough for many use cases in the pervasive computing domain.

Administering Smart Things: *How can protocols and patterns from the World Wide Web be utilized to create an infrastructure for the Internet of Things that is able to administer the magnitude of potentially integrated smart devices?*

In densely populated smart environments, it will get increasingly difficult for machines and human users to *find*, *select*, and *utilize* services that are provided by smart things in a fast, reliable, and user-friendly way. The task of finding relevant smart things is significantly more complicated than searching for documents, not only because smart things should be identified according to dynamic, contextual information but also due to the lack of a uniform way of describing the things, their properties, and the services they provide: a smart thing does not necessarily express its functionality in a way that traditional search engines – being geared toward finding textual documents – can process. To facilitate the search for smart devices and their services, we propose a scalable, Web-based infrastructure that simplifies the interconnection of heterogeneous embedded devices and is optimized for managing large numbers of interacting smart things.

Collaborating Smart Things: *How can physical mashups that involve multiple services from different devices be created by end users?*

Whenever an individual smart thing is not able to fulfill the user's requirements by itself, it can in principle collaborate with other smart objects and services that could help achieve the user's goal. For this to succeed, smart things must have access to information about what tasks other devices and services can achieve, and how this functionality can be invoked by them to fulfill a higher-level user goal. To enable this, we propose a method of outfitting smart things with functional semantic metadata that enables a semantic reasoner to automatically create composite services by combining information and functionality from multiple smart devices.

1.3 Thesis Outline

This thesis is structured as follows: In Chapter 2, we discuss how smart things that we consider throughout this thesis are connected to each other and how they communicate with clients using well-known and proven mechanisms and patterns from the World Wide Web. Based on these principles, we propose in Chapter 3 an approach to embed metadata into smart things' representations that enables user interaction devices such as smartphones, tablets, or wearable devices (e.g., smartglasses and smartwatches) to render suitable user interfaces for the direct interaction with smart things. We show that, by describing the high-level semantics of an interaction, our proposed model-based interface description scheme is expressive yet easily producible for creators of interface descriptions and that it supports the generation of intuitive modality-independent user interfaces. To initiate the interaction with a specific smart thing, we propose to use state-of-the-art visual object recognition technologies for identifying devices. The integration of computer vision methods with our user interface description scheme is discussed in Chapter 4, where we also discuss other approaches to the selection and identification of smart devices.

In Chapter 5, we move from considering the interaction with individual devices to user interactions with entire smart environments and present a system that enables humans to perceive communication flows within a smart environment in real time using visual object recognition technologies in combination with an augmented reality interface. Next, in Chapter 6, we introduce a Web-based management infrastructure for smart things that facilitates their discovery and look-up within densely populated smart environments. This chapter also includes a discussion of the metadata that the discovery component of our infrastructure tries to obtain by analyzing the Web representation of a smart thing when encountering the device for the first time. Finally, in Chapter 7, we demonstrate how users can be enabled to not only control individual, isolated smart devices, but also to manage compositions of services that are available in their smart environment – ideally, a user's smart environment would be aware of which high-level user goals could be achieved if all devices and services in the environment collaborated with each other, and would make this information available to the user. We discuss two approaches of how we attempt to provide this functionality: Our first method relies on developers who publish the relevant information and thus represents a crowd-based approach to service composition. As a

second approach, we propose to embed functional semantic metadata into services, which can then be used by a semantic reasoner to deduce service compositions at runtime.

To demonstrate the different systems for facilitating the interaction with smart things that we present in this thesis in the context of a real-world use case, we discuss how our software supports users when interacting with Web-enabled automobiles in Chapter 8. We show that our proposed user interface descriptions, their integration with object recognition technologies, the visualization of device communications, and the embedding of semantic metadata are indeed useful to support users when interacting with Web-connected cars. We conclude and highlight avenues for further research in Chapter 9.

CHAPTER 2

Connecting Smart Things to the Web

To enable convenient interaction with smart things requires solutions to challenges on several layers of the “interaction stack:” prior to addressing the *selection* of smart objects to interact with, the *generation of adequate interfaces* that facilitate the interaction with them, and the *automatic collaboration* of devices in smart environments, smart things must be enabled to communicate seamlessly with each other. This “Device Accessibility Layer” [73] forms the basis of all approaches to smart things interaction that are presented in this thesis – the objective of this chapter is thus to describe how devices and services that we consider are connected to each other, and how people can access them using interaction devices.

After giving an overview of the IP-enablement of smart devices, we discuss their integration on the application layer and the main architectural style that we adopt for modeling smart things and the services they provide. Within this discussion, we cover many properties of devices and services that we consider throughout this thesis and that will be frequently referred to in the subsequent chapters.

2.1 Convergence in the Internet of Things

Much of the current effort to interconnect more and more everyday devices is carried out under the umbrella of the Internet of Things vision. The core idea of the IoT is that the Internet shall be extended into the real world and include content that is generated by sensors, and by applications that in turn use the data and functionality these offer to provide higher-level services. Smart things – digitally enhanced, communication-capable objects such as wireless sensor nodes, mobile phones, and home appliances – could, for instance, be networked together to create environmental monitoring applications [240]. In the IoT, each smart thing is given its own IP address [52], so as to be able to communicate and interact with other things and services. By opening up new modes of interaction among things, the development toward the IoT will enable us to monitor the real world in real time by collecting up-to-date information directly from networked physical objects, and also control these devices remotely.

While the IoT makes it possible that smart things and the services they provide are reachable via the Internet, it does not directly target convergence aspects in smart environments – the IoT often requires humans and machines that want to make use of the capabilities of a smart device to have access to specialized client software and software libraries: the development of applications that integrate the functionality of multiple smart things consequently remains a challenging task because it requires expert knowledge of each involved platform on many layers, ranging from rather low-level know-how of embedded systems to high-level user interface design [190]. One major risk of failing to overcome this convergence challenge is that “Intranets of Things” are created that are not interoperable but rather form isolated islands on the application layer [73]. This development has in the past been particularly visible in the sensor networks domain [196]: today, such networks of spatially distributed sensor nodes are gradually being integrated with the Internet, a development that represents a major opportunity for traditional sensor networks applications as the usage of novel Internet- and Web-based interaction and management schemes for distributed sensor networks facilitates their deployment and operation.

2.2 The Web of Things

Unlike the IoT, where the *network-level* connectivity plays the central role, targeting the convergence of smart things on the *application layer* and facilitating IP-enabled devices from different manufacturers to cooperate with each other is the prime goal of the *Web of Things (WoT)* development. This goal, proponents of the WoT argue, can be achieved by exposing device application programming interfaces (APIs) that are simple to use for end users and by making Web-enabled smart things accessible for humans as well as other devices via standard software (e.g., the Web browser) and widely deployed protocols and standards (e.g., the Hypertext Transfer Protocol, HTTP) [77, 78]. Ultimately, smart objects shall become “first-class citizens” of the World Wide Web and therefore findable, usable, and shareable like any other hyperlinked resource [78] – devices can be connected using traditional Web concepts such as Uniform Resource Identifiers (URIs, [287]) and hyperlinks, and many mechanisms and patterns that make the Web scalable and successful are directly inherited by smart devices (e.g., caching, load balancing, and the stateless nature of the HTTP protocol). Fully leveraging the Web’s architectural principles and patterns fosters device interoperability and scalability while facilitating user interaction and openness. Having the entire interaction with a smart thing happen via Web protocols furthermore eliminates many compatibility issues that could otherwise occur due to vendor-specific protocols, and also lowers the barrier of entry for end users [176]. Finally, the WoT allows to use traditional Web services – in particular Web search engines – in conjunction with physical devices, thereby extending their utility beyond their traditional functions, and, in the case of Web search engines, allowing humans and machines to “query the real world” [136, 167, 197].

Today, tiny Web servers can already be embedded in many devices [50, 52, 53, 89, 240] that are, for instance, based on building-management technologies such as KNX

or enOcean [24], and – with the help of so-called *smart gateways* – the WoT can even extend to devices that are not IP-enabled [78, 227]. Integrating devices in the WoT should, however, also be possible in a way that is lightweight enough to be applicable to resource-constrained devices,¹ a requirement that lies at the heart of the Constrained Application Protocol (CoAP) which is currently in the final phase of being standardized by the Internet Engineering Task Force (IETF) [257]. This train of thought culminates in the idea of the “Thin Server Architecture” [107] that advocates that smart objects such as home appliances merely expose Web APIs to their basic components (i.e., sensors and actuators) and that most of the program logic of the devices and their services be deferred to remote servers.

2.3 Representational State Transfer

The smart devices that we consider in this thesis and that we strive to make simple to interact with for human users all form part of the WoT. In this section, we review the main underlying patterns and principles of the World Wide Web and, consequently, the WoT, because these form the architectural basis of the interaction mechanisms that we discuss in the subsequent chapters.

A large part of the success of the World Wide Web stems from its scalable architecture, generic interfaces, and loosely coupled components. In its idealized form, we refer to the basic architectural style that the Web adheres to as *Representational State Transfer (REST)* [56]. In REST, the primary abstraction of objects that provide information and functionality are *resources* that are identified in a uniform way using URIs (this is commonly referred to as the *Resource-oriented Architecture, RoA*). These objects can be queried and manipulated using a limited and fixed set of verbs (in HTTP, these are *GET*, *PUT*, *OPTIONS*, etc.) that have generally understood semantics (e.g., for HTTP, *GET* is considered to be free of side effects). Messages that are exchanged between client and server are *self-descriptive* and their structure is considered *common knowledge*, which is supported by a content negotiation mechanism used to determine the concrete resource *representation* that is transmitted between communication endpoints.

Fully subscribing to RoA and REST distinguishes the WoT from traditional WS-* Web Services [176] in that it does not use HTTP only as a transport protocol to convey specification-conform data, but rather directly exposes the functionality of smart things by adopting a REST-driven, resource-oriented architecture [189]. Although HTTP was designed as an application protocol with particular focus on scalability, many Web applications use it only as a transport protocol and therefore only utilize a fraction of its functionality: for instance, Web applications that rely upon WS-* technologies use only the HTTP *POST* operation to perform API calls on URI-identified endpoints and do not expose the manipulated resources themselves. Practices like these prevent applications

¹By this, we refer to hardware that falls into the categories *Class 1* or *Class 2* according to the IETF (see [106]): *Class 1* devices are capable of directly connecting to the Internet but cannot operate a full HTTP protocol stack and thus require lightweight and energy-efficient alternatives. *Class 2* devices almost exhibit the characteristics of “unconstrained” hardware, but can still benefit from lightweight protocols.

from taking full advantage of the Web architecture because they neglect the semantics of the interaction verbs: for instance, a **GET** could be used to signal that a specific interaction is free of side effects and therefore cacheable.

2.3.1 Basic REST Concepts

The REST architectural style is defined in terms of a set of constraints on the server interface and the client-server interaction. The goal when formulating these constraints was to explain what properties of the Web are responsible for its desirable qualities such as performance, scalability, simplicity, and modifiability [57]. In this section, we review the most important properties of systems that conform to the REST constraints (we refer to such systems as being *RESTful*) [189]. Although all of the concepts introduced here are crucial for the REST architectural style, we emphasize selected properties that are of particular interest in the context of the interaction with Web-enabled smart things.

Any RESTful system is based on a client-server paradigm, where requests are sent by clients to a server that processes them and returns a response. The main factor that distinguishes RESTful architectures from other client-server systems is that the interaction is centered around *resources* that reside on servers: clients can retrieve resource representations and modify the state of server resources by using these entities. This property is what gives REST its name: the client sends a request to the server whenever it is ready to perform a transition of a server resource to a new state, by changing the representation of that resource. The definition of what a resource is has seen multiple amendments during the history of the Web. Resources were originally defined to be “anything that has an identity” by the IETF in the year 1998 [286]² – that definition also included several examples for resources (an image, a service, etc.) and the notion that a resource does not have to be network-retrievable: for instance, human beings can also be resources. The currently valid definition of a resource was produced by the IETF in the year 2005 and forms part of the specification of Uniform Resource Identifiers (URIs) [287]: this document defines resources as “whatever might be identified by a URI,” and extended the scope of URIs to abstract concepts such as relationship types or numeric values. The WoT explicitly treats real-world objects as Web resources, including their physical parts (e.g., an actuator of a device) and virtual components (e.g., the currently measured value of a sensor). Sensor values can, for instance, be obtained from a device by using HTTP **GET** requests and actuators can be controlled by setting their state using HTTP **PUT**.

Apart from the client-server paradigm, the other REST constraints are *statelessness*, *caching*, *layering*, the *code-on-demand* constraint, and a constraint that is especially crucial when considering the interaction of humans and machines with Web servers: the *uniform interface*. REST requires that the communication between client and server must be *stateless*, meaning that a client request must include all information necessary to process it. The client can thus not directly use (possibly temporary) aspects of the server context that are not addressable by a URI. Statelessness governs how servers can

²Already in the year 1994, the IETF published an informational document that implicitly defines resources as anything that can be addressed [283].

manage recoveries from failure and the migration of states between machines, and is thus crucial for achieving fault-tolerance and load-balancing within RESTful systems. The *cache constraint* determines that clients in principle are able to cache server responses and that servers must specify whether their responses are cacheable (in HTTP, this is done using the **Cache-Control** header field). This constraint thus helps to avoid redundant requests to servers, which improves the scalability and performance of the system. The *layering constraint* targets the scalability and modifiability of a REST system: it emphasizes the necessity to reduce client-server coupling via information hiding, thereby also supporting caching as it should remain opaque to the client whether a request has been served directly by the addressed server or by an intermediary that has cached an earlier server response. The *code-on-demand constraint* specifies that clients can be temporarily customized by downloading and executing mobile code (i.e., client-side scripts). This improves the extensibility of REST systems because it reduces the number of features that need to be implemented client-side, but impedes the self-descriptive nature of requests, and was therefore defined as an “optional” REST constraint.

2.3.2 The REST Uniform Interface

Finally, the REST architectural style specifies a set of constraints on the client-server *interaction* itself by dictating that all communication between clients and servers must take place through a uniform interface. The four uniform interface constraints – identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state – are fundamental to RESTful systems because they allow to decouple clients and servers, thus enabling both parts of a client-server application to evolve independently. In HTTP, for instance, servers only process a finite set of commands with defined semantics (**GET**, **POST**, **PUT**, etc.; these are referred to as the HTTP “verbs”) – because server interfaces allow only these verbs, servers and client browsers can evolve independently: it is not necessary to update the browser for accommodating updates on a particular website.

REST mandates that individual resources be *identified* in requests, for instance using URIs – the usage of uniform identifiers for resources enables clients and servers to transfer information about the resource relevant to a request without pre-negotiated agreements. Whenever such information is passed in the context of a client-server interaction, it is in the form of a specific *representation of the resource* that can be parsed and manipulated locally by the client. The corresponding resource on the server is often modified or deleted during the onset of the interaction. To support the client when interpreting a resource representation – and the server when processing client requests – REST requires messages that are exchanged in either direction to be *self-descriptive*. One major mechanism to satisfy this constraint in HTTP is the usage of common Internet media types,³ as well as binding registration procedures for such types that have been defined by the IETF [295]. Arbitrating between different representations of a resource in HTTP is the task of its

³Originally, these media types were conceived as a common way of referring to non-ASCII parts of email messages. For this reason, their definition is part of the IETF Multipurpose Internet Mail Extensions (MIME) specification [284, 285].

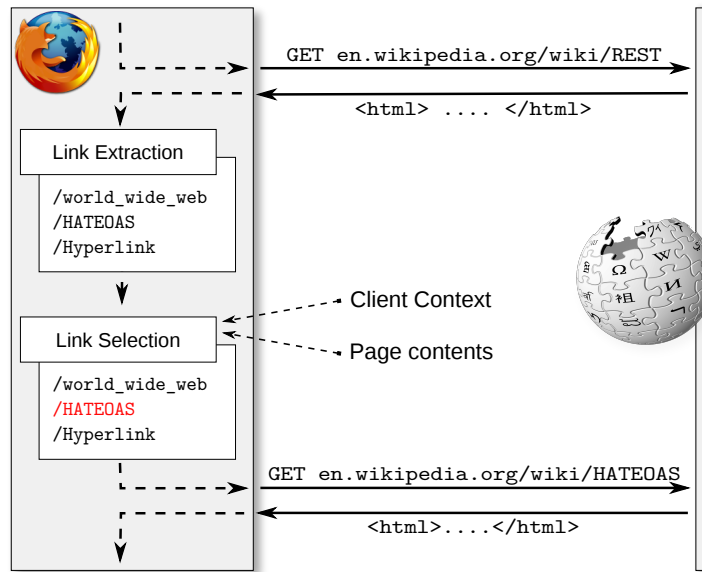


Figure 2.1: Illustration of the HATEOAS principle: users interact with Wikipedia by requesting an initial resource and interpreting its HTML representation. Hypermedia controls are provided by the resource and rendered by browsers in a way that makes them apparent as hyperlinks. Users select one of these depending on their context (e.g., personal interests) and the page contents (e.g., an advertisement) and invoke the matching hypermedia control.

content negotiation mechanism – for this thesis, this is particularly interesting as it allows to serve different representations of a resource (which represents a smart thing or one of its services) depending on whether the client that wishes to interact with it is a human or a machine.

The final and – in the context of this thesis – most important uniform interface constraint is the *hypermedia as the engine of application state (HATEOAS)* constraint. In a system that respects HATEOAS, clients can only perform state transitions on resources using actions that are provided within the hypermedia that the server delivers, for instance using hyperlinks. The core idea behind HATEOAS is that no knowledge about a Web application should be implemented in Web clients other than the ability to follow hypermedia links. This approach is central to avoid tight coupling between clients and servers: the client code can remain static even when the server code changes, because it is the task of the server to guide client actions by providing appropriate links for the client to interpret and follow.⁴ In short, as illustrated by Fig. 2.1, the HATEOAS principle puts all possible state transitions in a Web application under the control of the server.

At this point, it is interesting to note that most Web browsers actually break the HATEOAS constraint: the *Back* and *Refresh* buttons, standard components in all widely used browser implementations, provide to users the ability to re-send their most current request – and several earlier requests – on the click of a button. The reason why usage of these buttons often breaks Web applications (often in the context of e-commerce applications or booking systems) is precisely because they provide a way to circumvent HATEOAS: the server is no longer in control of the interaction.

⁴Colloquially, this principle is referred to as “follow your nose” [272].

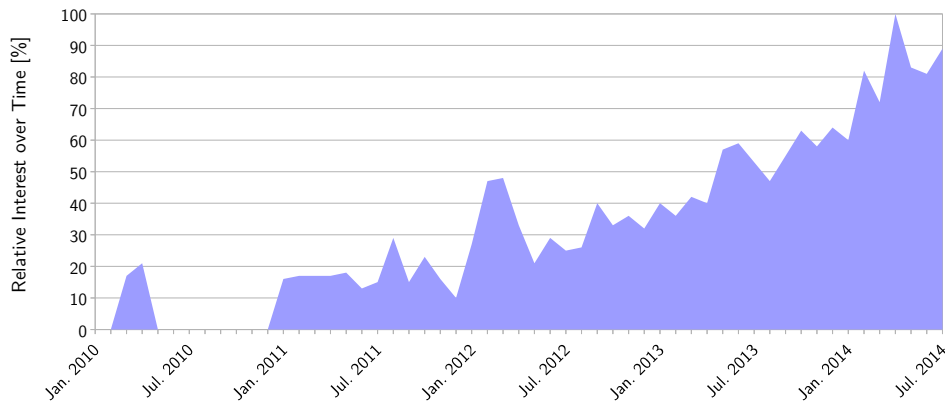


Figure 2.2: Relative interest over time for the search term “HATEOAS” (Apr. 2014 = 100) from Google Trends [278].

Having been widely neglected for a long time and often dismissed as a purely academic concept, the HATEOAS constraint has recently been attracting more attention by a broader audience including industry, as can be approximated by doing a Google Trends analysis of the term “HATEOAS” over the last four years (see Fig. 2.2). We believe that this rise in interest in this particular constraint is mainly driven by increasing demand for machine clients that automatically or semi-automatically use Web applications on behalf of human users. As is exemplified in the above example with the *Back* and *Refresh* buttons, humans are able to recover from failures that arise from the client breaking the HATEOAS contract – machines, however, in most cases are not able to react in an adaptive way to such failures.

To summarize, HATEOAS is what adds flexibility to REST systems. It is what allows Web applications and Web browser implementations to evolve independently, because browsers are merely general-purpose tools that provide to users the ability of following links in hypermedia documents: clients discover links at runtime and will therefore automatically adapt if links change, given that they have a way of interpreting what a specific link signifies. While humans certainly are able to do this, we will discuss several options for implementing such smart behavior for machine clients in Chapter 7, and propose a way of how machines can interpret and use functionality that is provided by services within ubiquitous computing scenarios. In our opinion, applying the uniform interface constraints, and in particular the HATEOAS constraint, to Web applications that are provided by physical devices represents one of the core advancements of the WoT over generic IoT systems. Uniform interfaces are crucial for achieving convergence between smart things, for enabling the orchestration of smart things, and for facilitating the interaction of human users with environments that are populated by them.

2.3.3 REST and WS-*

The main alternative to adopting the REST architectural style as an application layer for physical devices is using more traditional protocols and standards that are referred to

collectively as *Web services* or, in short, *WS-**. The main difference between these and REST is that, for *WS-**, URIs are used to denote *service invocations* themselves – this is the reason why *WS-** and similar approaches are said to implement a service-oriented architecture, or SOA.

The main technologies of the vast *WS-** protocol stack are SOAP⁵ for providing service access, the Web Services Description Language (WSDL) for describing service capabilities, and the Universal Description Discovery and Integration (UDDI) initiative for the discovery of services – other *WS-** standards define more meta-services such as addressing and security. At the time these technologies were conceived (the years 1995-2000), their purpose was to make services on the Internet discoverable and usable for clients in a language-independent way, since the APIs of service offerings on the Internet had at the time become very heterogeneous, for instance with respect to the format in which they consumed data and delivered their results.

The novel requirements of applications in the IoT, where services are usually not provided by enterprise servers but by – often resource-constrained – devices such as mobile phones or sensor platforms, necessitate that the suitability of the two approaches be assessed in light of their deployment on devices with rather limited capabilities [76]. Although *WS-** applications have successfully been deployed on resource-constrained devices [184], widespread consensus has been established in recent years that the REST architectural style is better suited for typical IoT applications, for reasons of higher performance on constrained devices [240] and better scalability [78], as well as better usability for programmers [76] – more lightweight forms of *WS-** services, such as the Devices Profile for Web Services (DPWS) [312], have come and disappeared again due to a lack of interest in these technologies by relevant parties mainly in the industry. Furthermore, given the ubiquitous usage of Web browsers and the fact that people are already used to exploring the Web using a browser, we can safely assume that REST-based systems are also simpler to access for end users than Web services. Also due to the discontinuation of *WS-** systems by major companies,⁶ REST, which was only a few years ago merely considered for purposes of “tactical, ad-hoc integration over the Web” and not for “professional enterprise application integration scenarios” [174, 176], is today increasingly seen as the de-facto standard for device integration when using Web protocols.

In our own work [76], we investigated how developers assess the two styles when being confronted with the task of learning both technologies and using them in the context of a mobile phone application that retrieves sensor data, both using a REST API and a system based on *WS-**. Our study among 69 novice developers in the year 2010 was motivated by the assumption that APIs that are easy to learn and use are necessary when applications are to be developed by a broad community [43] – as is the case at the moment with the IoT, where companies increasingly rely on external developers to build innovative services, and, thus, create added value for end users. Our study revealed highly significant differences in the perceived ease and speed of learning of the two technologies (see Fig. 2.3): for instance, 70% rated REST *easy* or *very easy* to learn while under 10%

⁵The abbreviation originates from the “Simple Object Access Protocol.”

⁶UDDI was discontinued by IBM, Microsoft, and SAP in the year 2006, and the functionality was removed from the Windows Server operating system in 2010. Google discontinued its *WS-** APIs in 2011.

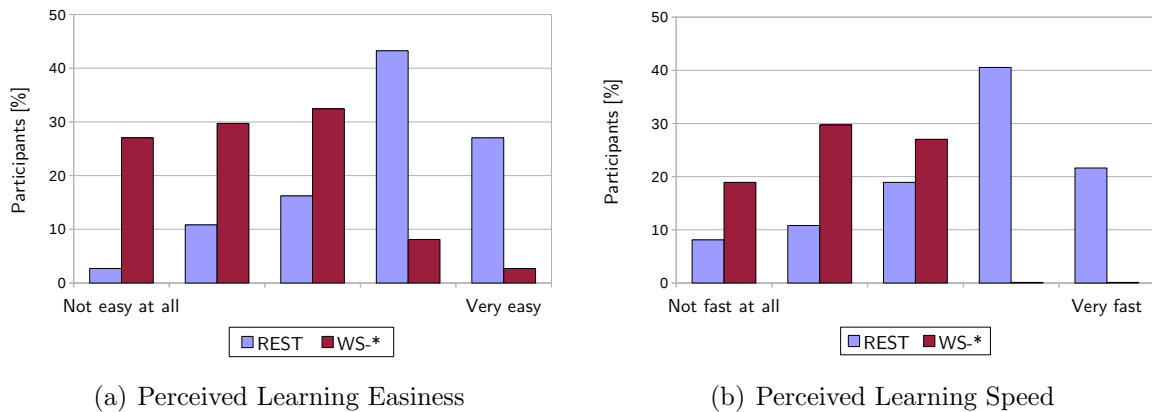


Figure 2.3: Results from our study to elicit the developers' perspective on REST and WS-*. Participants rated the speed and ease of learning on a 5 point Likert scale (1=Not fast/easy at all ... 5=Very fast/easy). The results show that REST was perceived as being easier (a) and faster (b) to learn than WS-*.

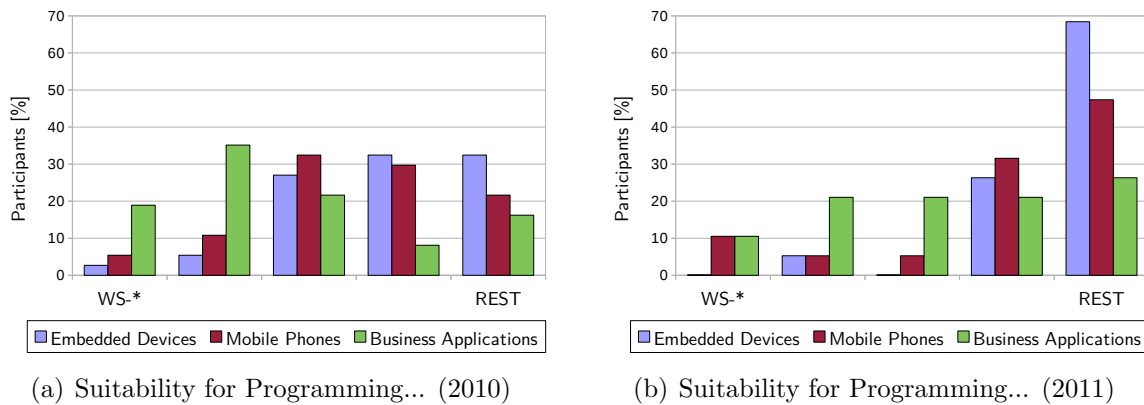


Figure 2.4: Results from our study to elicit the developers' perspective on REST and WS-*. (a) Participants believe that REST is better suited for IoT applications that involve mobile and embedded devices. (b) Results from a replication of the study in the year 2011 indicate higher preference for REST in all three application domains.

said the same about WS-*. The results also show that REST was significantly preferred for applications that involve mobile and embedded devices (see Fig. 2.4(a)), while the preference of WS-* for “business applications” was not statistically significant. When we repeated the study in the year 2011, we discovered that the preference for REST was now even stronger across all three application domains (see Fig. 2.4(b)).

2.4 Summary

In this chapter, we introduced the vision of the Internet of Things, an Internet that extends into the physical world and will provide us with huge amounts of real-time data and functionality. In our opinion, the IoT represents the next logical step in the evo-

lution of the ever more dynamically evolving Internet that has recently undergone its last pronounced transformation, toward enabling faster and more integrated collaboration between humans: the IoT will do the same for physical devices. We expect that this development will generate substantial added value for consumers by allowing for new modes of interaction between static documents, services, smart things, places, and people. The IoT also enables the real-time management of business processes, for instance in logistics, and will allow businesses to rapidly react and adapt to events in the physical world, for instance in industrial manufacturing.

After an introduction to the Web of Things, we discussed two possible architectures to achieve coherence in the IoT on the application layer: REST and WS-*. We showed that, while others established that REST provides higher scalability and performance than WS-* on resource-constrained devices, our results from a study among novice developers show that REST also stands out as the preferred architecture from a developer's standpoint. We discussed the REST architectural style in greater detail, as we will refer to specific properties of such architectures – specifically the uniform interface and HATEOAS – in later chapters of this thesis: In the next chapter, we show how REST supports the embedding of information about the interaction capabilities of a smart thing and about user interfaces that are suitable to control it, directly in its Web resources. In Chapter 6, we present an implementation of an IoT management infrastructure that is based on REST, and show that REST features can be leveraged to accelerate searching for services that are provided by smart things in such an infrastructure. Finally, in Chapter 7, we discuss the integration of service offerings across multiple smart devices and services in smart environments – there, we show to what extent the HATEOAS constraint enables the creation of collaborative applications, where it fails to do so, and what other technologies are available to offset its shortcomings in this domain.

CHAPTER 3

User Interfaces for Smart Things *

Once a smart device is integrated into the Web of Things and provides a Web API to access it, humans and machine clients alike can in principle invoke its services via ordinary Web requests. Because the smart things we consider are additionally based on the REST principles and feature a resource-oriented architecture, we can assume that each basic functionality that a device provides – for instance, accessing its measured sensor values or controlling its actuators – is embodied in the form of a Web resource. Humans can thus directly access and control a smart thing using a Web browser, a widely available tool that most users are familiar with [15, 39]. In this case, all user interaction typically happens via the human-readable Web representation of the smart device (i.e., HTML, in most cases) that displays sensed values and provides a simplistic, form-based, interface to control actuation of the smart thing (Fig. 3.1(a)). While such an interface is easy to deploy or can even be generated automatically, it is often neither intuitive nor efficiently usable and the interaction itself may be cumbersome, especially when using a mobile device to interact with the smart thing. As a remedy, interfaces may be provided that are manually tailored to specific smart things (Fig. 3.1(b)). However, these usually are expensive to create and not flexible enough to adapt to different platforms and scenarios.

To bring Web-enabled smart things into peoples' homes and to their workplaces and enable humans to better interact with smart environments, more intuitive but still easily deployable interaction mechanisms are required. In this chapter, our focus is on supporting *explicit human interaction* with smart things which emphasizes the direct and immediate monitoring and control of such devices – a concept different from smart environments providing *invisible* background assistance. Such immediate interaction is relevant especially if the controlled devices provide information or perform actions of immediate value to the user (e.g., monitoring electricity meters or controlling multimedia systems).

To enable the *automatic generation* of user interfaces for smart devices, a model-based approach seems to be best suited [63, 161]: The smart thing embeds a description of how other devices can interact with it in the form of a User Interface Description Language

*Parts of this chapter have been published in *ACM Transactions on Computer-Human Interaction* 21 (2) (2014), as Mayer, S., Tschöfen, A., Dey, A.K., Mattern, F.; User Interfaces for Smart Things – A Generative Approach with Semantic Interaction Descriptions. [143]

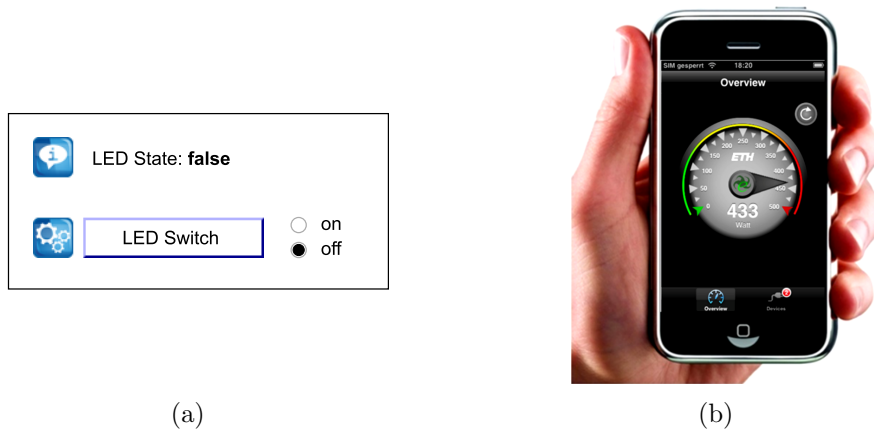


Figure 3.1: (a) Browser-based interface to toggle a Web-enabled LED [135]. (b) Manually tailored mobile user interface of a smart electricity meter [236].

(UIDL), and interaction devices (e.g., remote controls or smartphones) use this information to provide an appropriate concrete interface to the user. Such a system enables plug-and-play interaction within smart environments with no configuration effort other than embedding the appropriate descriptions. Thereby, it greatly reduces the amount of time and work needed to create appropriate and easily usable user interfaces for smart devices [161] – this is crucial when targeting the direct and immediate interaction of human users with devices in the WoT.

In this chapter, we present a high-level description scheme for smart things that captures the *semantics of an interaction* with the device rather than providing an explicit, concrete encoding of an appropriate type of user interface or its appearance. Based on this approach, we propose a modality-independent taxonomy of interaction semantics for monitoring and controlling devices. We furthermore present a concrete description language that captures interaction semantics and, based on this language, a prototype implementation of a universal remote control application for smartphones. This application as well as our description scheme have been evaluated within a controlled laboratory environment, with mock-up smart devices, in deployments in private homes, and in a study that targeted the understandability and simplicity of the proposed description scheme.

Our focus is to allow the provisioning of interaction descriptions for smart devices by adding a minimal amount of markup that is easy to understand and simple to produce for developers. Still, the description scheme is general enough to be applicable to a wide range of interaction use cases, where we primarily consider devices that monitor and control physical quantities in the real world. Such devices are widespread in home and building automation systems (e.g., light dimmers, window blind motors, or household appliances) but can also be found in cars (e.g., air conditioning), electric musical instruments, toys, and many other devices that we interact with in our daily lives. More and more, these traditionally simple, isolated devices are being equipped with processing and communication capabilities, thus transforming them into smart things. Since the main functionality that such devices offer is to sense and/or actuate the physical world, they can be modeled as actuators, sensors, or sensor-actuator-composites. A light dimmer, for

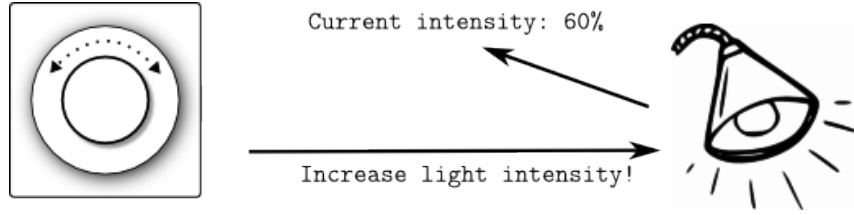


Figure 3.2: An interactor (light dimmer) and a stateful atomic interactive component (dimmable lamp) whose state can be queried and manipulated.

example, is a simple actuator that controls the electric power supply of a lamp and consequently its brightness. A toy robot, in contrast, might have multiple motors to control its movable parts, and sensors to perceive its environment.

After discussing our terminology for referring to the different components that are involved in an interaction with a smart device, we introduce our approach of describing the high-level semantics of interactions in Section 3.2. We detail how these interaction semantics can be captured in Section 3.3 and discuss elements of a language to describe them in Section 3.4. In Section 3.5 we show a prototype application that interprets our interaction descriptions and can be used as a generic mobile user interface in smart environments. We present an evaluation of our language with respect to its generality, usability for end users, and producibility for developers in Section 3.6 and discuss the positioning of our approach with respect to related work in Section 3.7.

3.1 Terminology

We consider an *atomic interactive component* to be a physical or virtual object that provides a specific functionality to the user and is not reasonably divisible (e.g., a light switch). In contrast, a more complex device such as a VCR can be subdivided into multiple atomic interactive components, for instance its *Play/Pause* buttons and its volume controller. An atomic interactive component either provides its current internal state as data (sensor) or performs an action when reacting to a command (actuator).

We differentiate between two types of actuators: A *stateful actuator* is an actuator whose state can, in addition to being manipulated by sending commands, also be queried. One example for this is a dimmable lamp (see Fig. 3.2) whose state is the lamp’s current intensity/brightness. A *stateless actuator* can be manipulated like a stateful actuator but does not hold a representation of its internal state: It can be triggered but not queried. An example is a digital doorbell that plays a sound when triggered by a button press. Every stateless actuator can in principle be transformed into a stateful actuator by exposing its current state. However, for this type of actuator, representing the state is often not necessary or results in too much overhead because the state does not play an important role or is hard to capture.

The device or software abstraction that a human uses to interact with an interactive component is called an *interactor*. In the example depicted in Fig. 3.2, this is the light dimmer knob. Interactors can take many forms ranging from traditional graphical user interfaces (GUIs) to gesture or speech-control interfaces and to physical buttons or knobs.

3.2 Interaction Semantics and Atomic Interactive Components

We postulate that an interaction description scheme should be usable by heterogeneous interaction devices, involving gesture-based, speech-based, graphical, or physical interfaces such as physical buttons or sliders [166]. Furthermore, to foster widespread adoption and actual usage in practice, a user interface modeling language should, in addition to being expressive enough, be easy to understand for developers [124]. Ideally, it should thus enable the embedding of interaction descriptions with only a few lines of easily producible markup. Approaches to providing UIDLs proposed to date are limited in their support for these requirements. Most target the provisioning of interface descriptions for complete devices (see Section 3.7 for a thorough discussion of related work): the user interface of a VCR, for instance, is usually specified in a single document that, apart from describing the interfaces to each of the VCR's components (e.g., the *Play*-button), lists all dependencies between components of the device. An example of such a dependency is that the *fast-forward* button of the VCR should only be active when a video is playing. Instead of describing devices as a whole in this way, we claim that especially within a Web context it is more beneficial to embed interaction information *directly into the devices' atomic functional components* and not explicitly specify such dependencies. Other projects in the domain of UIDLs have also advocated the decomposition of appliances into their atomic interactive components (e.g., XWeb [164], the URC standard [246], the PUC project [161], or MARIA [171]), but have not explored this further as an approach that could yield simpler yet expressive interface description languages.

The traditional way of describing how one can interact with atomic interactive components is to associate them with data type information: An element which has an *integer* or *float* type with a range can be graphically represented by a slider; an *enum* type corresponds to a drop-down menu, etc. We argue that providing such a *data model*, however, is only a specification of the *program interface* to an interactive component. While this enables rudimentary interaction with devices, the specification of data types is not sufficient for creating intuitive interfaces which, in our opinion, requires capturing the *semantics of the interaction*. We therefore propose to abstract from *user interface descriptions* to *interaction descriptions*, meaning that we do not model concrete interface elements but instead the semantics of the interaction of the user with a device (what exactly we mean by *interaction semantics* is discussed in detail in Section 3.3.2). The possibility of adding more abstract information about interface elements has also been expressed by the authors of the above-mentioned PUC papers and the URC standard, but has not been further explored as a possibility to simplify user interface descriptions.

When analyzing traditional (built-in or remote) user interfaces for devices such as the ones mentioned above, one notices that certain types of interactors (e.g., various kinds of knobs, combinations of buttons, etc.) occur again and again but control very heterogeneous types of actuators and sensors. For instance, from a user interface point of view, a light dimmer knob and a thermostat knob are equivalent: although certainly a little unusual, it would be possible to control a lamp's brightness with a thermostat knob.

This interchangeability is not confined to interactors with similar physical appearance. In fact, both brightness and temperature could also be controlled by a graphical slider widget, or with speech commands (“increase/decrease brightness/temperature”). The reason for this is that the *semantics* of interaction are the same for the two interactive components: In both cases, the user *scales a physical intensity*.

We argue that this observation can be generalized: in the following, we propose a classification of interactive components into semantic interaction categories which suggest appropriate interactors, but are still general enough to be applicable for a wide variety of smart things. Interestingly, although this idea originates from the analysis of physical actuators and sensors, it can also be applied to a range of appliances and software applications whose actuators and sensors are virtual, because they employ physical metaphors (e.g., switches or scroll bars) for their control. We hence take a different point of view on user interfaces: rather than considering a “knob” as a physical entity that is emulated in GUIs, we consider the interaction semantics of knobs and argue that such “conceptual knobs” occur in different forms which all encode the same interaction intention, namely *scaling a value*. We furthermore show that different interaction semantics do not exist by themselves but rather can be arranged in a semantic interaction hierarchy, with the most specific interaction types that carry most semantic constraints at the bottom and the most generic interaction types at the top.

Building on these concepts, we suggest that it is possible to create interfaces that represent a substantial improvement over traditional type-based widgets by implementing only a relatively low number of abstractions to capture interaction semantics. This can be leveraged to reduce the developers’ effort for providing interface descriptions and thus fosters their widespread adoption. Furthermore, specifying interaction descriptions on this more abstract level allows human interaction with smart things *regardless of the type of interaction device or the modality of the interaction* (e.g., gestures, haptics, or speech). Finally, the selected level of abstraction and the hierarchical structuring of interaction semantics enable the interaction device to *adapt the manifested interface* to its own capabilities and/or user preferences. For instance, an interactor without a graphical interface could still be able to generate an interface for switching between different modes, or for scaling (e.g., using its gyroscope). We thus claim that a scheme which describes interactions semantically is, in general, more suitable to support user interaction within smart things environments than other approaches to user interface modeling proposed in the literature (e.g., [161, 164, 171, 246]).

Restricting our scheme to the description of the atomic interactive components (i.e., in our case, URI-identified resources) of a smart thing leads to a low entry barrier for users (e.g., Web developers) to create interaction annotations for devices. Our approach is thus well-adapted to the devices that are our main concern: in the context of the WoT, smart things are rather simple devices whose capabilities (i.e., interactive components) are hierarchically structured due to the adoption of a hierarchical resource-oriented architecture (see Chapter 2). For this reason, it is possible to view such a device as a structured collection of its sensors and actuators rather than as a single entity that supports complicated tasks.

However, even if interactive components of a smart thing are described independently, they can still be aggregated within composite user interfaces *without explicitly modeling semantic relationships* between components. This is the case because the user interface should usually only *reflect* a devices' internal state and not introduce further constraints. As an example, consider a simple user interface for a television set that comprises a volume level and an on/off switch. Showing the volume level interface as inactive when the device is switched off is then not a constraint that is introduced by the user interface but rather reflects the fact that the volume cannot be set without switching on the television set. In our proposed concept, one would *independently* describe the switch and the volume level and leave the decision whether the volume level is active or not to the controlled device itself rather than to the interactor that renders the user interface. We thus argue that the proposed language can also be used to describe interfaces for composite devices with dependencies between components and refer to the discussion of this property in Section 3.6.

Apart from keeping descriptions simple to understand and produce, the decomposition of devices into their atomic interactive components has further advantages regarding the generation of user interfaces: while it remains possible to create *thing-centric interfaces* (i.e., to meld all capabilities of a device into a single interface), the proposed approach adds the possibility to create *task-centric interfaces* by integrating components of different smart things within a single user interface. Examples for interfaces that are tailored around a specific task are an interface that displays information from all temperature sensors within one building or an interface geared toward watching a movie that incorporates sensors and actuators from the television, stereo set, and DVD player. This property thus allows different devices or persons to view “their” smart environment from different perspectives.

3.3 Describing Interaction Semantics

In classical model-based approaches, interactive components are modeled mainly or exclusively using data types. While our proposed description methodology also makes use of data type information, it merely does so to describe the exchanged data and not the user interface of the component per se. In fact, we propose to describe components by using interaction descriptions that consist of two parts, *data type information* for the data exchanged with the component and information about the *high-level semantics* of the interaction.

3.3.1 Data Types

In the proposed interaction description format, every atomic interactive component has an associated data type which represents the type of the entity state for sensors and stateful actuators. For stateless actuators, it gives the type of the argument that should be supplied when triggering the actuator. Our description supports the data types *boolean*, *integer*, *number*, *enum*, and *string*, where the well-known types have the usual semantics.

enum requires the definition of allowed values, which can be given as a static list of values or as a dynamic list referenced by a URL. Optional arguments include defining the value's *unit* and the *allowed values* or *range*. Allowed values for the *string* type can be described as a regular expression or a definition according to the *JSON Schema* format [291]. Again, we stress that these types are not given to derive an appropriate user interface but rather to populate derived interfaces with meaningful values.

3.3.2 Semantic Interaction Abstractions

Knowledge about the data type already specifies how to interpret the component's state and which values are allowed as state and thus enables basic type-safe interaction. Considering as an example a window blind controller with the states “down,” “stop,” and “up,” a graphical interaction device could, for instance, generate a drop-down list using only the type information (in this case, *enum*). However, this interface is hard to understand and use, and unnatural: it demands that users perform the mapping of their interaction intent (moving the blind) to the choice of a state of the blind motor. Furthermore, to bring the blind down just a little bit, the user would have to open the drop-down list twice in rapid succession to start and stop blind movement, respectively. These problems arise because the data type is only a specification of the *program interface* to an interactive component and does not consider the semantics of the interaction. As a way of capturing these interaction semantics, we propose the concept of *semantic interaction categories* for interactive components which we call *interaction abstractions*. We have identified about a dozen distinctive interaction abstractions that capture interaction semantics related to sensing as well as stateless and stateful actuation.¹ Clearly, not every interaction with any smart thing falls into one of these categories and the classification should therefore be considered as a proof of concept that can be extended as required. However, we found that the proposed categories already cover all of the use cases that we encounter within our deployments. The set of abstractions was obtained by considering typical devices in several core domains where smart things play, or are supposed to play, an important role:

- Home and building automation systems: lighting, HVAC, curtain & blind control, audiovisual equipment, security, electricity metering, etc.
- Home and office appliances: washing machines, coffee machines, etc.
- Auditorium control systems: lighting, A/V selection and controls, controls for peripherals such as blackboards or projectors
- Cars: air conditioning, drive controls, comfort controls, etc.
- Public services: ticket machines, vending machines, etc.
- Electronic toys and musical instruments

Three of the proposed interaction abstractions apply to sensors, two to stateless actuators, and eight to stateful actuators. For sensors (Table 3.1), an interaction abstraction

¹Interestingly, a similar number of so-called *conceptual transitions* have been identified that describe dependencies and interactions between entities in *conceptual dependency theory* [204].

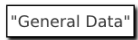
Name	Example Symbol	State Description	Example
get data		General data.	Display current song.
get value		Ordered domain.	Display temperature.
get proportion		Ordered domain with fixed range.	Display load of a server.

Table 3.1: Interaction abstractions for sensors.

Name	Example Symbol	Description	Example
trigger		Trigger an action.	Reset button.
goto		Adjust one-dimensional state.	Change track on hi-fi unit.

Table 3.2: Interaction abstractions for stateless actuators.

captures the nature of the measured data. The `get proportion` abstraction, for instance, suggests that the value measured by a sensor should be considered with respect to its possible range and rendered appropriately, for instance as a progress bar or gauge. For actuators (Tables 3.2 and 3.3), the interaction abstraction stands for a primitive of actuation which provides information with respect to three dimensions: the *abstraction of actuation*, the *semantics of values* in the domain of the actuator, and the suggested *interaction pattern*. As an example, consider the `move` abstraction (see Table 3.3) with data type *enum* for the window blind controller that was described before:

- The *abstraction of actuation* refers to the physical (or metaphorically physical) actuation that the actuator can perform. The `move` interaction primitive, for instance, implies that the performed actuation refers to a movement.
- The *semantics of values* associate the values in the domain of the interactive component with a meaning or function. For `move`, the domain is ordered with a neutral value in the middle, the neutral value corresponds to no movement, and values below and above correspond to a movement in one or the other direction.
- Concerning the suggested *interaction pattern*, a `move` interaction is usually composed of two parts: starting and stopping the movement. An interactor can implement this by, for instance, falling back to the neutral value (*stop*) when the user activity ends (e.g., when the user releases the corresponding button).

Some interactors that satisfy these requirements for the `move` abstraction are shown in Fig. 3.3 on page 26. Other components with the interaction abstraction `move` are, for

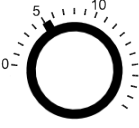
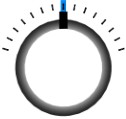
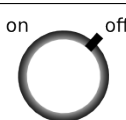
Name	Example Symbol	State Description	Example
set		General data.	Set text displayed on a screen.
set value		Ordered domain.	Set minutes until alarm.
level		Ordered domain with a neutral value.	Scale an image.
set intensity		Ordered domain with fixed range.	Set loudspeaker volume.
switch mode		Operating mode.	Switch ventilation mode.
switch		On or off.	Switch on/off lamp.
position		Point in one-dimensional space.	Position window blind in one-dimensional space.
move		One-dimensional movement.	Move window blind.

Table 3.3: Interaction abstractions for stateful actuators.

example, a robot arm motor, or an actuator that supports rewinding and fast-forwarding a video (by “moving” the current point in time). As Fig. 3.3 suggests, interaction abstractions are modality-independent and can be mapped to graphical widgets, physical interactors, and speech or gesture commands. Furthermore, a particular interaction abstraction usually can be superimposed on interactive components of multiple different data types. This is the case because it encodes the high-level semantics of an interaction, while the data type depends on the implementation of an interactive component. As an example, a more sophisticated blind control could be of type *integer* (instead of *enum*), where the absolute value of the state would correspond to the speed of the blind movement. Still, the appropriate *interaction abstraction* for this controller is **move**.

The interaction abstractions can be organized in hierarchical taxonomies (Fig. 3.4), where the root abstractions **get data**, **trigger**, and **set** are the most abstract ones possible. These three abstractions yield simple, solely type-based interactors. Descending within the hierarchy, the semantic information gets more concrete. As an example, consider **level**, the parent abstraction of **move**. It suggests that the domain of the actuator’s

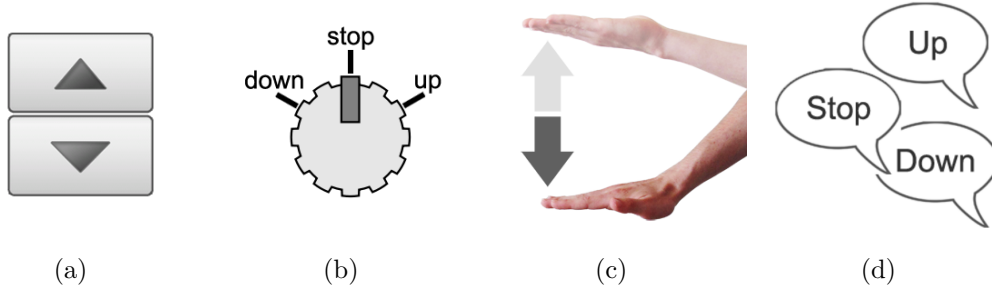


Figure 3.3: Example user interfaces for (vertical) move actuators. (a) and (b) represent graphical interfaces, (c) a gesture-based interface and (d) a speech interface.

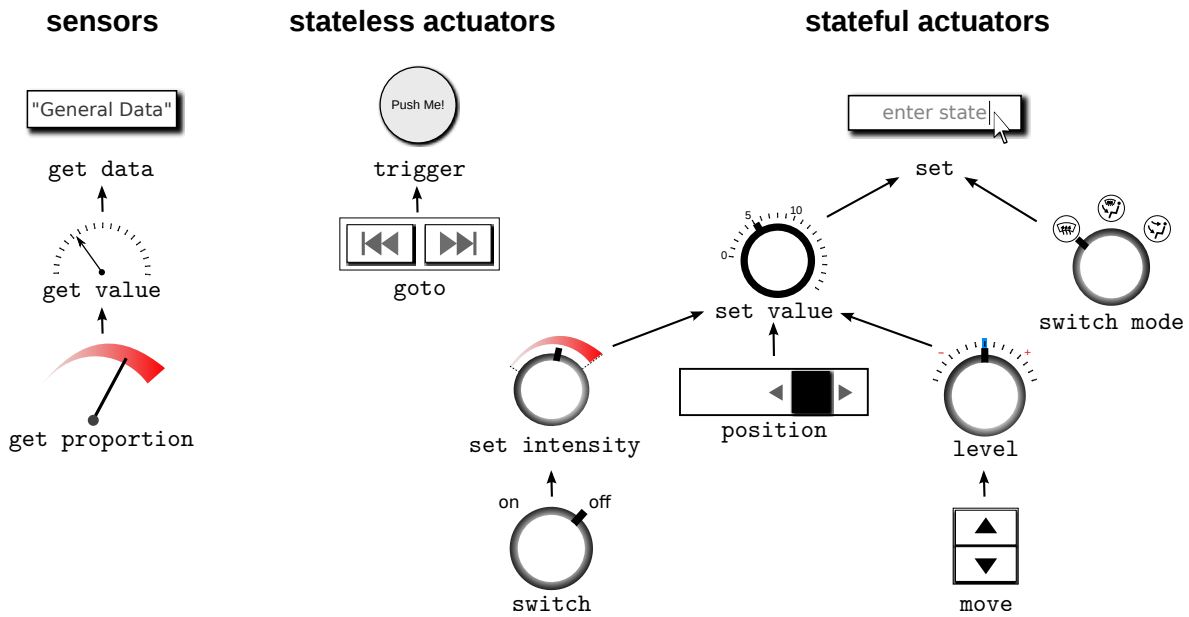


Figure 3.4: Interaction taxonomies for sensors, stateless actuators, and stateful actuators.

state is ordered with a neutral value which should be reflected by corresponding interfaces, e.g., a knob that snaps into place in the middle position. The *move* abstraction adds information about the effects of setting this state: a movement.

The hierarchical organization of interaction abstractions can be exploited as a fallback mechanism: if an interaction device does not know or does not support a particular interaction abstraction, it can traverse the tree upward until it finds a more general abstraction that it can handle. This enables the scheme to also be used by simple interactors that only support a subset of the available abstractions. Furthermore, the taxonomy stays extensible as new abstractions do not have to be known by all interaction devices from the beginning. One can thus build large taxonomies with arbitrarily specialized interaction abstractions at their leaves.

Within our implementation of this concept, we aimed at generating small taxonomies to make the descriptions easy to understand, embed, and interpret for developers. We found, however, that the small set of abstractions proposed above already covers a large set of interactive components (see Section 3.6).

3.4 Elements of a Semantic Interface Description Language

The classification of interaction semantics together with the described taxonomy can be used to define a concrete semantic interface description language by embedding information about the appropriate data type and interaction abstraction as metadata into the interactive components of devices. In this section, we present elements of an implementation of such a language that is based on the interaction abstractions shown above (Tables 3.1-3.3), and that allows user interface devices to render interactors when confronted with a corresponding interactive component. This language was used in a prototype implementation of a generic mobile user interface for smart things which is described in Section 3.5 below. For reasons of legibility and understandability, the reference implementation of our proposed language is shown in JavaScript Object Notation (JSON) [262]. A description of the window blind controller mentioned above, for instance, then looks as shown in Listing 3.1.

The proposed concepts of our language can, however, be expressed and embedded in multiple formats, for instance as XML documents or as HTML-based Microformats markup. For annotating Web-based smart things, we found HTML Microdata [331] to be particularly convenient, because it allows to embed information that is meaningful for humans and machines within the same document. Using Microdata thus allows us to create a document that looks like an ordinary website to humans, but still contains all information that is necessary for interaction devices to generate user interfaces (see Listing 3.2). An alternative to this direct embedding of interaction metadata in the Web representations of our smart things is to store the descriptions on a remote server and have devices provide links that point to them, for instance using Web Linking [292] – this is particularly useful for annotating resource-constrained smart objects.

```

1 {
2   "type" : {
3     "name" : "enum",
4     "values" : ["down","stop","up"]
5   },
6   "abstraction" : {
7     "name" : "move",
8     "orientation": "vertical"
9   }
10 }
```

Listing 3.1: Semantic interface description of a window blind controller in JSON format.

```

1 To <span itemprop="name">move</span> the blinds <span itemprop="orientation"
   ">vertical</span>ly, set the controller to one of the values <span
   itemprop="type-range">[down,stop,up]</span> (data type: <span itemprop
   ="type-name">enum</span>).
```

Listing 3.2: Semantic interface description of a window blind controller as Microdata markup.

```

1 {
2   "id":"ch.ethz.inf.vs.wot.ui.move",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the move interaction abstraction: temporarily
6     level a value of a device property negatively or positively. Implies a
7     virtual or physical movement.",
8   "properties":{
9     "abstraction": {
10       "type":"object",
11       "required":true,
12       "properties":{
13         "name": { "type":"string", "enum":[ "move" ] },
14         "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors",
15         "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.neutral",
16         "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.orientation"
17       }
18     },
19     "$ref":"ch.ethz.inf.vs.wot.ui.type"
20   }
21 }

```

Listing 3.3: JSON Schema document that describes the move interaction abstraction.

Our proposed interaction description language has been built around a small kernel that consists of data type and interaction semantics information to keep descriptions simple to understand and write for humans, and easy to parse for machines. To enhance the experience of using an interface created from these descriptors, we additionally propose *optional* properties that allow for the refinement of the interaction abstraction, for instance, to specify details of the actuation or to define dedicated values. We briefly describe two such properties that we found to be particularly helpful for augmenting automatically generated interfaces, *anchors* and *orientation*. *Anchors* can be used to add special meaning to certain values or value ranges. For instance, specific values (e.g., the range of 95-100% when displaying the load of a server) can be marked as potentially harmful (or particularly desirable) which then can be reflected in the user interface. Similarly, especially for the *move* and *position* abstractions, the desired *interface orientation* can be specified, for instance, as *vertical* for a window blind controller. Additional properties could be defined for increased customization of user interfaces – doing so excessively, however, could potentially corrupt the language’s simplicity. To allow for the possibility of manually tailored user interfaces and the mixing of these and automatically generated interfaces, our proposed description scheme also includes the possibility of specifying links to dedicated Web interfaces that can be displayed by interaction devices.

We used JSON Schema, a specification used to define the structure of JSON data, to create a formal definition of our language. This definition includes a human-readable documentation of the language and contains all information necessary for structural validation of interaction descriptions which is useful in automated testing. To give an example of such a specification, JSON Schema document in Listing 3.3 shows the structural defi-

inition of the `move` abstraction. The schemas that are required to validate an interaction description using this document (e.g., *anchors* or *type*) are defined in separate definitions. These, as well as schema documents for all other interaction abstractions, can be found in Appendix A of this thesis.

3.5 A Generic Mobile User Interface for Smart Things

To evaluate the discussed concepts in practice, we implemented a prototype application for mobile devices running the Android operating system. This application interprets our interaction descriptions and allows end users to interact with devices via automatically generated interfaces. End users can also store interfaces locally on their interaction device and can aggregate multiple of these within composite interfaces (i.e., as widget lists). Specifically, using the application, users can create new task-centric composite interfaces and give them a name that one can better relate to, such as “My Lecture Hall Controls.” To populate a composite interface with widgets, users can then select from the stored interactors. Composite interfaces can also be associated to specific locations: for instance, an individually tailored lecture room interface could be loaded whenever the user enters that room. Operation within secured environments is enabled by prompting the user to enter a username and a password to interact with restricted device components.

Our interactive components embed interaction descriptions as Microdata within their HTML representations, which is mapped to our JSON reference format using a discovery service that can handle smart things with embedded semantic descriptions (this service is discussed in detail in Chapter 6). When the mobile application discovers such a component, it retrieves its interaction description and instantiates an appropriate interactor. If multiple suitable interactors are found, one of them is rendered and the user is given the

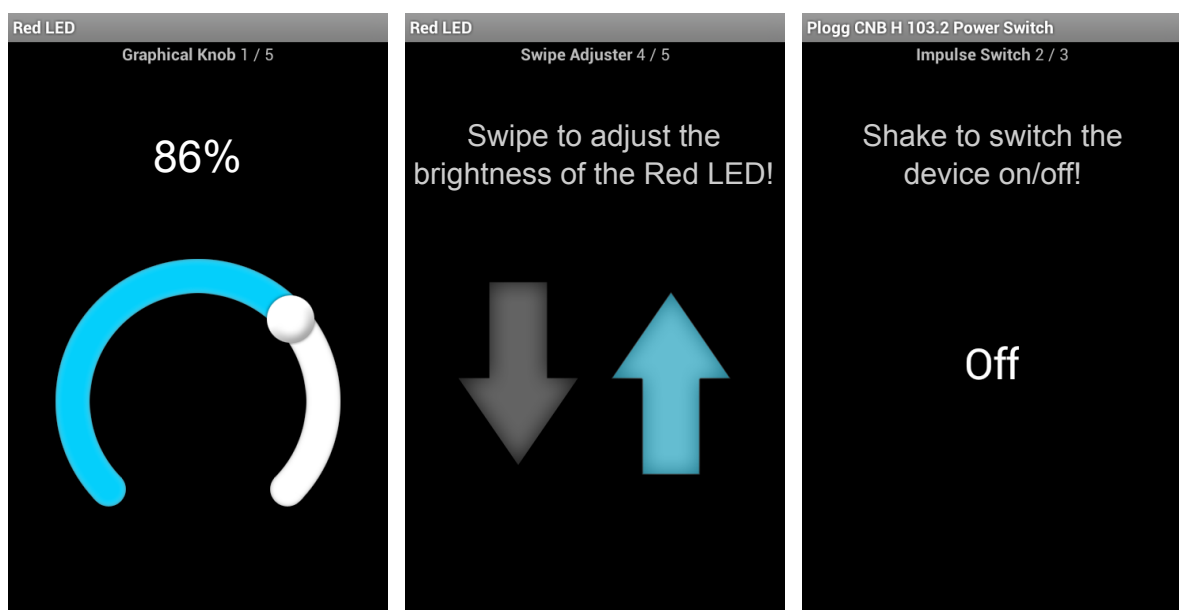


Figure 3.5: Interfaces for controlling the brightness of a LED (“set intensity”, left and middle) and a power switch (“switch”, right).

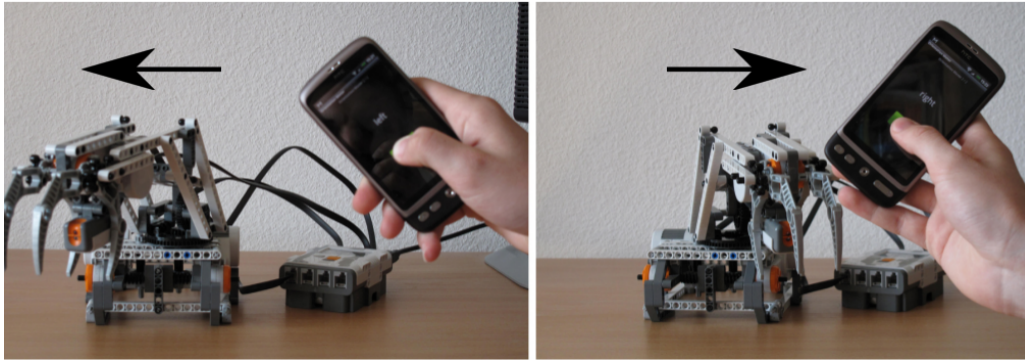


Figure 3.6: Rotating the smartphone to move a toy robot arm (“move”).

opportunity to browse through the other interactors. From this point on, the application uses HTTP requests to get and update the state of the component by exchanging plain values that correspond to the type definition of the interactive component. Our application immediately provides multiple different interactors to users because the prototype was created to load interfaces with different interaction modalities for demonstration purposes. For a final system that is used by end users, a single interactor could be rendered, with the possibility of loading more interfaces if desired by the user.

We implemented various graphical interactors (e.g., gauges, click wheels, knobs) that correspond to the defined abstractions (see Fig. 3.5 for some examples). Some of these also capture the optional definitions (for instance, a knob with a value range as *anchor* that causes vibration and turns red when this range is entered). Furthermore, we used smartphones as haptic input devices and mapped interaction abstractions to physical movements of the handset or interaction with the touchscreen. For instance, the handset can be tilted or turned like a knob to switch between operation modes or to move a robot arm (see Fig. 3.6). One can **trigger** or **switch** by shaking the handset, and by swiping over the screen, one can use the **goto** abstraction for stateless actuators. The implemented generic mobile user interface for smart things shows that it is indeed possible to map the interaction abstractions to heterogeneous, also nongraphical, modes of interaction.

For further facilitating the interaction with smart devices for end users, we ported our application to user interface devices beyond smartphones, in particular to personal wearables such as smartwatches (see Fig. 3.7) and smartglasses. The motivation for considering wearable devices was to “reduce the time between intention and action” [215], i.e., the time that the user requires for taking his mobile phone out of the pocket and activating the application, in our case.

To further explore the modality-independence of the proposed description scheme, we also investigated *speech-based* interaction where appropriate speech commands are inferred based solely on the semantics captured by the interaction abstraction. The only additional information needed to enable actuation commands is related to the *selection* of the interaction target (e.g., by using the target’s name): for a volume controller, the target could be “Volume”, for a window blind motor, “Blind”, and so on. This information is not captured by the interaction abstractions since it is specific to every single actuator and determined by user preferences. It can thus be provided either by the interactive



Figure 3.7: Interfaces for controlling the volume of a stereo set on a smartwatch (“set intensity”).

component (e.g., as an additional *name*-annotation) or alternatively by giving the user appropriate means to name smart devices himself. We defined speech commands such as “move *target* up/down” and “stop *target*” for vertical move actuators. Note that, besides leveraging the semantic information of the interaction abstractions to choose appropriate command phrases to listen for, we can also benefit from associated information about the interaction pattern. For instance, the pattern in a *move* interaction is to start the movement and then stop it again when the moved object has reached the desired position. For speech interaction, this pattern translates to listening for a “move *target* up/down” command, followed by a “stop *target*” command rather than only a single command, such as, for instance, for a *switch*. Finally, the automatically created speech interfaces also allow users to select their desired mode of interaction, for instance, by choosing which of the commands “increase/decrease *target*” or “set *target* to *value*” to use for *set value* interactors. We implemented speech-based interfaces for all proposed interaction abstractions in our application and found that, in all cases, these were appropriate and easy to use for controlling smart devices.

To start controlling a smart device, the user interface device must initiate a connection with the device and retrieve the embedded information about its interface. Because our prototype application deals with smart things in a Web context, we assume interactive components to have a Uniform Resource Locator (URL). Given such URLs for smart things, resource association in our mobile prototype application can be performed by various means: Apart from manually entering the URL of the object to interact with, the application is able to decode 2D barcodes that encode device URLs. Furthermore, when installed on a device that features an active near field communication (NFC) component, an appropriate interactor is displayed automatically when the phone comes close to an

NFC tag that encodes a resource URL. By making use of geographical location information offered by many of our prototype devices, we also added context-sensitive behavior to the application: using the GPS module of the mobile device, interfaces that are stored on it are ranked with respect to their distance to the user and the closest interfaces are directly presented in the application's main interface. Finally, we explored how smart devices could be selected by the help of current image recognition technologies – we discuss this in greater detail in Chapter 4.

3.6 Evaluation

The proposed description language and the mobile prototype application have been deployed in various environments to determine whether our approach is general enough to capture the interaction semantics of typical devices in Internet of Things scenarios. We demonstrate the generality of the language by discussing the multitude of devices that were annotated using the language and give details about the concrete deployments in our laboratory and in private homes in Section 3.6.1. Next, in Section 3.6.2, we demonstrate that our proposed interaction descriptions are producible not only by experts who are already familiar with our system but also by individuals with little prior training. To show this, we conducted a user study among 780 students from all faculties of our institution whose task was to create interaction descriptions for 19 scenarios from the home automation domain. This study thus elicits the developers' perspective with respect to the creation of user interface descriptions. Finally, in Section 3.6.3, we show that the interfaces that are generated from our descriptions can be efficiently used by end users. After discussing how users interacted with annotated devices from our deployments, we elaborate on two case studies: The first shows how our language can be used in the context of a lecture hall control system. Here, we particularly consider the creation of composite, task-centric, user interfaces. The second case study, a user interface for a music player, demonstrates that our interaction descriptions can be used to create user interfaces for devices with a complex internal state, even though they do not explicitly model dependencies between interactive components of a device.

3.6.1 Assessing Generality: Laboratory and Real-World Deployment

To assess the generality of our proposed language, we first deployed the system in a laboratory environment. We added interaction annotations to several existing deployments of smart things present in our lab: SunSPOT sensor nodes (sensors: temperature, light, acceleration, orientation; actuators: tri-colored LEDs), Ploggs electricity meters (sensor: electricity meter; actuator: power switch), a toy robot (sensor: ambient light; actuators: 3 motors), a remote-control toy car, mobile loudspeakers, and a smart thermostat. In addition, we created mock-up implementations of a home automation system with embedded interaction annotations (lighting, blinds, stereo set, TV) and a lecture hall control system (volume and microphone controls, lighting control, A/V, peripherals, etc.). Specifically for the lecture controls, we modeled the exact capabilities of the system installed in our

institution's lecture halls, which had not been considered earlier when designing the interaction abstractions. To evaluate the performance of the scheme with respect to more complex devices, we also created a Web proxy for the iTunes music player application and modeled interfaces to its functionality (e.g., controlling playback, adjusting the volume, choosing a track to play from a playlist) using our interactions markup. Both, the lecture hall controls and the music player, are discussed in more detail in Section 3.6.3.

Our scheme and the prototype application were also tested outside of a laboratory setting, as two members of our research group deployed the system in their private homes. These individuals have used the system about once per day, to control entertainment equipment and smart electricity outlets with metering and switching capabilities. At the time of writing, one of the private deployments still exists and has been running for a year (intermittently). We found that our proposed interaction descriptions covered all sensors and actuators present in our laboratory deployment, in the home automation scenarios, the lecture hall control mock-up, and the music player.

3.6.2 Assessing Producibility: User Study

Apart from exploring whether our language is suitable for describing user interfaces in our use case scenarios, we also investigated the producibility of our proposed interaction description language. A preliminary evaluation showed that for members of our research group it was easy to create the interaction markup for typical use cases: after a two-minute introduction to the language, these individuals were able to apply the data type information and interaction abstractions to all devices and software components described above. However, to assess the accessibility of the description scheme and the ease of creating such interaction abstractions for individuals that had not been exposed to the system before, we conducted an online user study. Participants ($N = 780$) were asked to select appropriate interaction annotations and data types from the set described in Section 3.3 for 19 different scenarios ranging from a simple doorbell button and lighting scene controls to the description of interactive components of a VCR. Our test concentrates on the *selection* of appropriate abstractions and data types instead of having the participants *implement* a description because the implementation step can be fully automated by providing an application that takes the selected abstraction/data type pairs as input and produces annotations in the JSON reference format (see Section 3.4).

The study participants were students from all faculties of ETH Zurich with a self-reported average proficiency with information and communication technologies (ICT) of 3.65 ($SD = 0.82$) on a 5 point Likert scale (1=No Knowledge, 2=Basic Knowledge, 3=Good Knowledge, 4=Advanced Knowledge, 5=Expert Knowledge). The participants had no prior knowledge of our project and no training with using the interactions scheme. They were presented with a one-page description and reference document *during* the survey which they were asked to study for about 2 minutes before working on the scenarios. For every scenario, the participants were asked to complete four tasks: (1) Select the most appropriate interaction abstraction, (2) select the appropriate data type, (3) give a confidence level for their choice in (1), and (4) give a confidence level for their choice in (2), where they selected the confidence levels on a 5 point Likert scale (1=Not confident

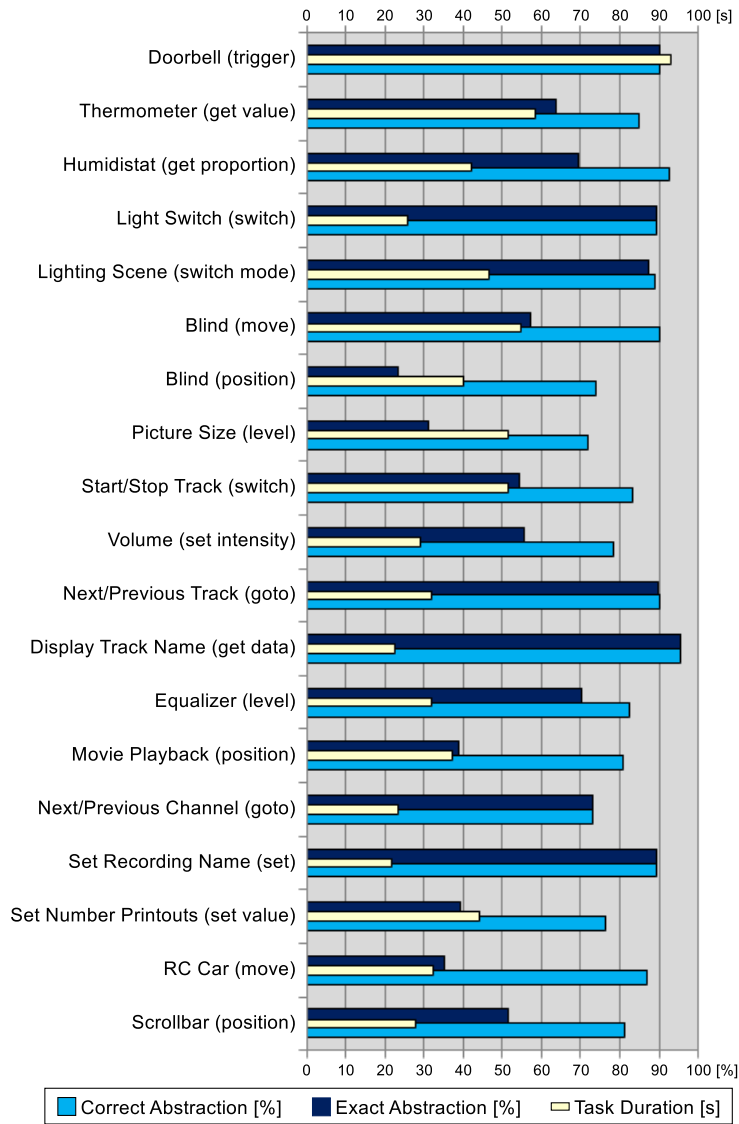


Figure 3.8: Performance and timing values for each of the 19 scenarios in our study of the producibility of the proposed interaction descriptions.

at all ... 5=Very confident). We also recorded the time taken by the participants to work on the individual scenarios.

Across all scenarios, the participants selected a correct interaction abstraction in 84.2% ($SD = 6.99\%$) of the cases on average. To do all four tasks associated with a scenario, they needed on average 40.3s, where the high standard deviation of 16.7s is due to a large part from the high average time of 93s that participants spent working on the first scenario (a doorbell button) and can thus be largely attributed to habituation effects, as the order of the scenarios was not randomized. The self-reported confidence level of the participants was 4.09 on average, with minor fluctuation across all scenarios ($SD = 0.26$). Evaluating the performance of participants with an ICT proficiency of (5/Expert Knowledge) ($N = 170$) separately revealed that these performed a little better than the average, selecting an appropriate abstraction in 88.7% ($SD = 10.47\%$) of the cases (16.9 correct answers out of 19).

Fig. 3.8 shows the average performance of the participants as well as their timing values

for each of the 19 scenarios. We present two distinct values to assess the participants' performance: the values for *Exact Abstraction* show how many of the study participants selected the interaction abstraction which captures best (in our opinion) the semantics of the described scenario, while the *Correct Abstraction* designates the percentage of participants who selected a sub-optimal abstraction that is appropriate for the scenario but captures less of the semantic information. For each scenario, the *Exact Abstraction* is stated in brackets. In general, the set of *Correct Abstractions* for a given scenario is the *Exact Abstraction* plus all abstractions on the path of the most appropriate abstraction to the tree root in the taxonomy (see Figure 3.4). From the data, one can see that, for some of the scenarios, people strongly agree with each other and with our assessment concerning the type of abstraction to be used (e.g., for the *Light Switch*, *Lighting Scene*, and *Display Track Name* tasks). For others, though, there is considerable disagreement about which abstraction of interaction to use. As an example, consider the *Picture Size* scenario, where participants were asked to specify the appropriate abstraction to set the zoom level for a digital picture frame between 20% and 200% where 100% is considered the neutral value of the interaction. In our opinion, the *Exact Abstraction* for this scenario is `level` as it allows the specification of 100% to be the neutral value of the interactive component. 31.2% of the participants indeed selected the `level` abstraction, however, another 39.1% chose to either model the interaction as `set intensity` or as `set value`. This is not wrong but rather represents a different way of interpreting this interaction which does not emphasize the modeling of a distinct neutral value. For the *Equalizer* scenario, where the *Exact Abstraction* is also `level` and it is more obvious that the neutral value should be explicitly modeled, agreement between participants is much higher: 70.3% of participants selected the `level` abstraction, in this case.

For other categories, such as `move` (scenarios *Blind (Move)* and *RC Car*), abstracting from the scenario to the appropriate interaction specifier and especially matching the states of the interactor (up-button pressed, down-button pressed, no button pressed) to the corresponding actuator states (up, down, stop) was more subtle and hard to grasp for the participants. Another interesting scenario is *Blind (Position)*: here, only 23.2% of the participants selected the `position` category while 44.9% selected its parent abstraction `set value` which demonstrates that they did not include the semantic information regarding the positioning in one dimension. A possible explanation for this behavior is that scrollbar-based window blind controls are not widespread and thus were considered unnatural by the study participants.

Because our interaction abstractions are modeled on “natural” types of interaction with devices and software abstractions, we did already expect that individuals would be able to annotate devices before seeing the results of the user study. However, we were still surprised by the high accuracy and high degree of agreement with our choices, and especially by the very low amount of time that participants required to produce the descriptions, which was under one minute per scenario in most cases. Summarizing, the results of our study show that the proposed scheme is very accessible, and not only for people with good knowledge of ICT systems: most participants were able to productively use it within minutes and with only negligible prior training. Considering the timing

and confidence values, the tasks were also fast and easy to perform. We expect the discrepancies between the choice of optimal and sub-optimal abstractions to strongly decrease when individuals approach the task of modeling more rigorously and in the context of *actually* providing user interfaces rather than answering questions in a survey.

3.6.3 Assessing Usability: Deployments and Case Studies

After discussing the generality of our language and its producibility for developers, we discuss the usability of interfaces that are created from our descriptions in this section. In Section 3.5, we introduced a concrete implementation of a mobile application that generates interfaces for all defined interaction abstractions and enables the user to create composite, task-centric, interfaces. We want to point out that, due to our interaction descriptions being language- and device-independent, this application represents only one way of interpreting our interaction abstractions, where the interaction abstractions were mapped to simple Android widgets or made use of the Android API for sensor access and haptic feedback. Our prototype application was tested by several members of our research group and used to control and monitor the devices described above.

Test subjects reported that the generated interactors felt intuitive and appropriate for controlling and monitoring all devices. Specifically, giving only very little information (i.e., only the data type and name of the interaction abstraction without any of the optional properties) in most cases was sufficient for creating an intuitive user interface, as our descriptions consider the high-level semantics of interactions on multiple levels, as detailed in Section 3.3.2. Test subjects especially enjoyed those interactors that bridged multiple modalities by, for instance, making use of the sensors of the mobile device (e.g., shaking the phone or speech input). When we enriched the descriptions with some of the defined optional description elements, test subjects in particular liked the haptic feedback capabilities of the prototype application (e.g., vibrations and sounds triggered by *anchor* annotations).

The usability of the interfaces that are generated by our specific prototype is, however, grounded in the usability of the underlying Android widgets and therefore does not allow to conclude that our description language necessarily leads to usable and intuitive user interfaces in all cases. Still, our prototype shows that the language can definitely be used to create good user interfaces, even though only simple mappings between our abstractions and Android widgets are employed. One could also implement device controllers that map our descriptions to different, more customized, final interfaces, or to interfaces that support even more modalities, such as gesture-based interaction. Furthermore, applications could be created that allow for more sophisticated composite user interfaces – our prototype uses rather simple widget lists for this purpose.

In the following, we discuss two case studies to illustrate different aspects of our language and the prototype application: The first targets personalized composite user interfaces in the context of a lecture hall control system. The second demonstrates that our language can indeed be used to describe devices with a complex internal state, although the interaction descriptions do not specify dependencies between different interactive components of a device.

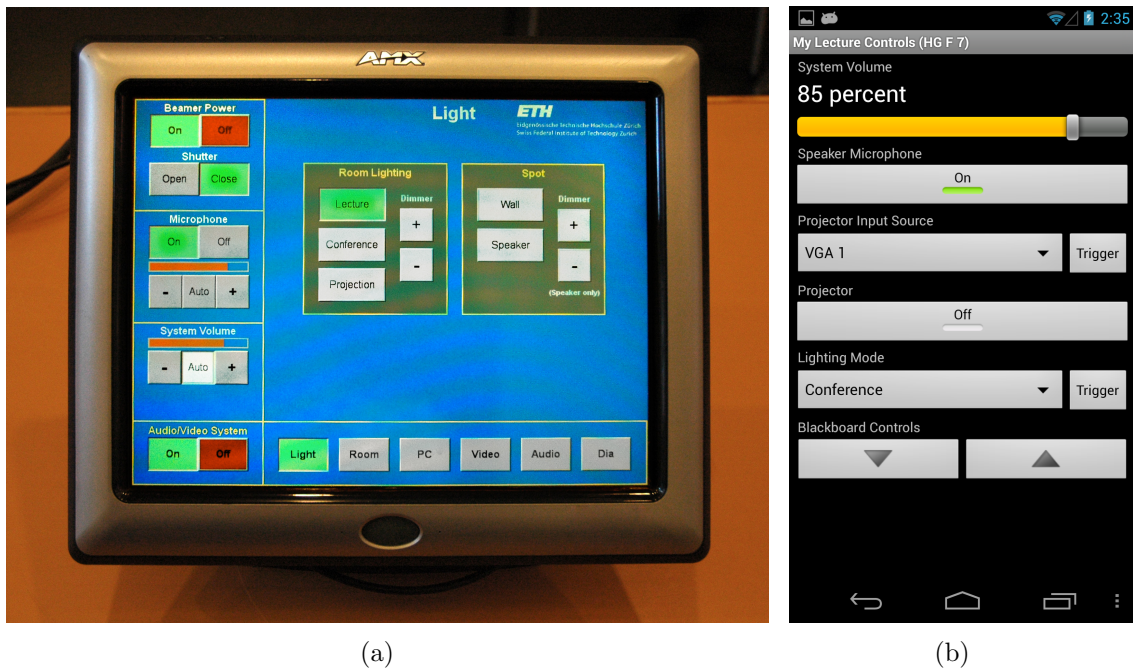


Figure 3.9: (a) Picture of the lecture hall control system in use at our institution. (b) Individually composed task-centric interface for accessing frequently used controls, rendered on an Android smartphone.

3.6.3.1 Case Study: Lecture Hall Controls

To test our approach within a real-world setting, we created a mock-up that emulates the specific auditorium controls that are in use in our institution² (see Fig. 3.9(a)). From past experience, we know that users who were not familiar with this system had trouble navigating its different tabs (*Lights*, *Room*, etc.) in search of their desired controls, a common mistake being that the *Video* tab was selected when looking for the controls to select the system’s video input source. We also experienced that non-German speaking individuals had trouble with some of the interface labels due to poor translation, for example with the “Beamer Power” controls (top left corner of the system in Fig. 3.9(a); the term “Beamer” is commonly used in Germany to denote a “Video Projector”). Our goal was to recreate all atomic interactive components of this lecture hall control system and let users configure composite interfaces that are customized according to their individual preferences using our prototype application. We had not considered this specific auditorium control system, or lecture room controls in general, when creating the interaction abstraction categories and designing our language.

To test our idea, we created a mock-up lecture room automation back-end, added all interactive components of the auditorium control system as endpoints to that server, and described them using Microdata annotations. In total, we needed less than half an hour to annotate all 28 components, which include the room lighting (setting the lighting level and mode), shades and blinds controls, ventilation mode settings, and controls for the video and slide projectors, sound system, and blackboards. Some of these components

²An AMX LLC Level 3 Modero Auditorium Control System.

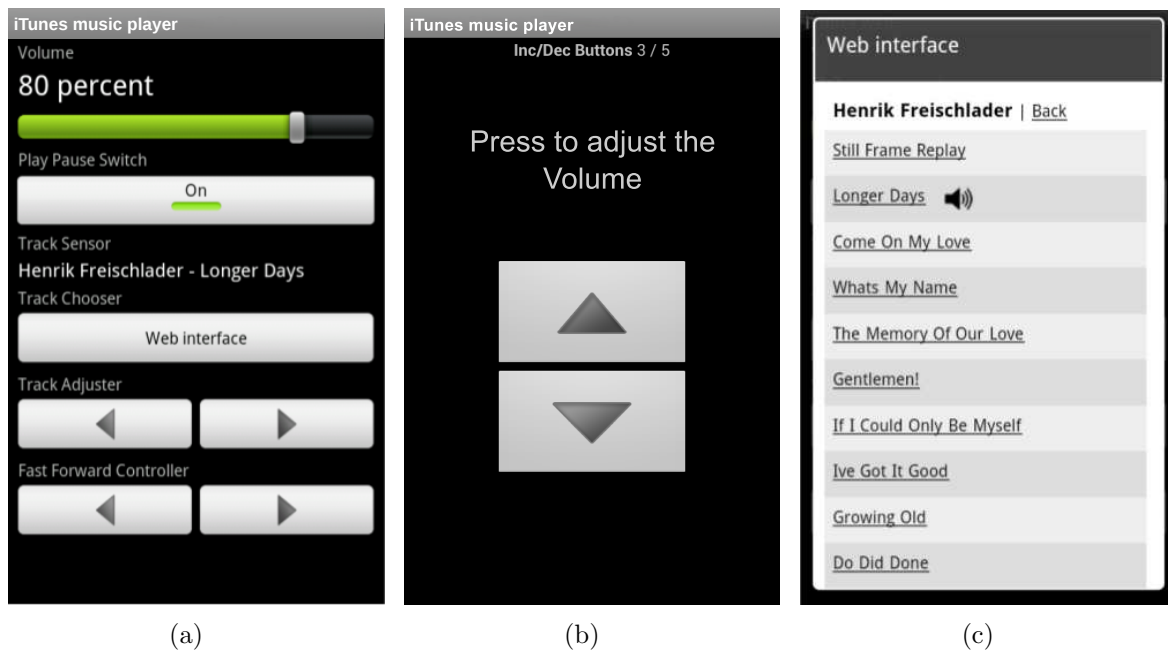


Figure 3.10: Example user interfaces for the iTunes application, rendered on an Android smartphone: (a) Composite interface that displays several interactors associated with the iTunes application. (b) Full-screen user interface for the volume controller (“move”). (c) Manually created Web interface to select songs.

have strong dependencies between each other: for instance, the projector shutter is only available when the projector itself is switched on.

The composite interface that results from combining a specific user’s most frequently used controls is shown in Fig. 3.9(b). To create that interface, one uses our prototype application to access the URL of the mock-up (e.g., by scanning a barcode or an NFC tag that is attached to the physical device, or to a proxy) which causes it to load all interface descriptions of the individual components and store the corresponding interfaces locally. He then instantiates a new task-centric interface, gives it a name (in Fig. 3.9(b), this is “My Lecture Controls (HG F 7)”), and selects which of the 28 loaded components this composite interface should contain. The user can also choose to interact with each of the components individually by accessing the corresponding “full-screen interactor” (some examples of these were shown in Fig. 3.5 on page 29). Components can also be rearranged, and can be removed from a composite interface.

3.6.3.2 Case Study: Music Player

Our description language does not allow to explicitly model logical dependencies *between* interactive components of a device. This means that it cannot be used to express how different resources that all belong to the same smart thing influence each other and how they affect the global state of the device. For instance, and referring to the example shown in Section 3.2, switching a television set off using its on/off button has an immediate consequence on the internal state of the device (i.e., it is now switched off) and also affects its other interactors: it is not anymore possible to control the volume of the TV, or

change the channel. Although such dependencies cannot be modeled in our system, it can still handle complex devices whose components are tightly coupled: using our generated interfaces to interact with the iTunes application – a smart thing with strong dependencies between its actuators and sensors and a complex internal state – is intuitive and effective (see Fig. 3.10). Although only the smart thing itself (i.e., the iTunes application) keeps the global application state and delivers only partial views to the interaction application on the smartphone, the generated interface for the music player creates the illusion of the mobile application being aware of the full application state. This is possible because the interaction between the smart thing and the smartphone application is, indeed, two-way: the interaction application can permanently update the state of the rendered interactors, by querying the smart thing. Therefore, if, for instance, a specific interaction becomes impossible (such as controlling the volume of the TV set in the example above), the user interface can immediately reflect that change. Example interactive components that we modeled for the iTunes application are its volume control (`set intensity`), Play/Pause switch (`switch`), Rewind/Fast Forward (`move`), an interface to skip tracks (`goto`), and the current track name display (`get data`).

3.7 Related Work

Model-based user interfaces have been investigated for a long time, initially with the goal of relieving application programmers from the task of manual GUI creation. [100] and [159] represent examples of early work that focused on the automatic generation of GUIs. This investigation led to the emergence of several model-based user interface description frameworks (see [160] for an overview and classification). However, the automatically generated interfaces were often not well adapted to the application which led to poor user experience [158] and the process of creating the models themselves proved to be rather cumbersome [161].

The Personal Universal Controller (PUC) / Pebbles project represents a pivotal step in the development of automatic user interface generation that would transform handheld computers into universal control devices [161, 162]. There, a description concept and concrete UIDL was proposed for appliances such as televisions, telephones, VCRs, and photocopiers. The proposed UIDL focuses on enabling high-quality interfaces, where some specifications consist of as many as 100 functional elements. The authors emphasized the producibility of interface descriptions and thus have designed the language to be easy to learn and use. This has been verified in a user study where subjects needed only an hour and a half of studying a tutorial document to be able to write specifications for a VCR interface. In [163], the authors present an extension of their system and introduce “smart templates” such as *media-controls* or *time-duration*, to better encapsulate the *meaning* of interactors – the *media-controls* template, for instance, contains *Play* and *Stop* controls. In contrast to our work, appliances are thus viewed as collections of tightly coupled “appliance objects” and their descriptions explicitly include logical dependencies between these functional units. Other approaches to mitigate the problem of interface specifications that are hard to create include the development of specialized authoring

software to allow developers to produce user interface descriptions, which has been done, for instance, in the MARIA project [171].

The tight coupling of functional user interface components makes interface descriptions hard to produce and, as we have shown, is not necessary to create usable and intuitive interfaces. Rather, following our approach, only the atomic interactive components of a smart thing are described on a level that allows to exploit metaphors that developers are already familiar with (i.e., the example symbols that are associated with our interaction abstractions). This usage of meaningful abstractions to avoid dealing with low-level details indeed represents the main motivation for model-based approaches – by focusing on making our interaction metadata simple to understand, produce, and embed, we thus attempt to “overcome the traditional separation between end users and software developers” [170]. Furthermore, in our approach, the generated interfaces are understood as a representation of the internal state of the device which is continuously updated. [116] refers to such interfaces, which feature bidirectional communication between the controlled smart thing and the interactor, as *complementary*, *duplicated*, or *detached* user interfaces, depending on what kind of (attached) user interface the controlled device itself provides. This feature directly enables to transfer interfaces from one device to another at runtime in a process called *user interface migration* [170]. Due to the platform-independence of our language, this can involve migrating user interfaces between heterogeneous devices with diverse capabilities, for instance from a smartphone to a physical switch or button. Because our language specifies the high-level semantics of concrete interactions, it also supports multiple contexts of use and preserves usability under these adaptations, properties which are referred to as *multi-targeting* and *plasticity* [31].

Our work fully integrates multi-modal user interfaces for heterogeneous interactors, and thus differs from approaches such as PUC which are aimed at “traditional” mobile user interfaces with touch panels or small keys [222]. This is important as the rise of the *ubiquitous computing* paradigm – where a person uses multiple networked computing devices that are embedded in everyday real-world objects – has led to a broadening of the design space of automatic user interface creation [181]. In the *ubiquitous computing* paradigm, interactions often take place in a spontaneous, ad-hoc fashion. Content and user interfaces now have to be adapted to a wide variety of devices with varying screen sizes (e.g., mobile phones vs. public displays) and heterogeneous capabilities (e.g., networked physical buttons vs. tablets with touchscreen, gyroscope, and accelerometer). This has in turn led to user interface models being increasingly abstracted from concrete GUIs to place more focus on user interaction on more abstract levels which in particular targets multi-device interfaces [13, 170]. One of the first examples for this development is presented in [88], which introduces the notion of “universal interaction” and represents a first step in the development of a service architecture that supports heterogeneity in interactors and the controlled objects.

Another approach that targets the automatic provisioning of user interfaces that support multiple modalities of interaction is the XWeb project [164]. In XWeb, user interface information is conveyed using *XView* descriptions that contain information about interface elements such as icons, field names, layout, and help texts and can also be used to

generate speech interfaces. In that respect one should also mention Interplay [150], which focuses on device and content integration as well as on the interaction with devices at the level of the *task* to be accomplished. To this end, this work provides valuable insights about speech analysis to enable the system to map spoken commands to specific tasks. An ambitious project that focused on enabling people with disabilities to control everyday devices using their specialized controllers – for instance, user interfaces that are attached to wheelchairs – is the Universal Remote Console / V2 (URC) specification [246]. There, target devices (e.g., ATMs) transmit a description of their abstract input/output behavior to the controller which then renders an appropriate user interface. Finally, the SUPPLE project [63] addresses the provisioning of alternative user interfaces by stating interface generation as a discrete constrained optimization problem that can be solved on the fly, where, for instance, a person’s motor impairments are modeled as a cost function to guide the optimization. This paper furthermore includes a discussion of some other approaches to model-based user interface generation that have been published in the last decade. In contrast to our approach, the projects mentioned in this paragraph require user interface descriptions that are difficult to create for users or, especially in the case of SUPPLE, can only be created by experts.

With respect to related work in the domain of model-based user interface descriptions that discusses various abstraction levels of user interface design [149, 171], the main novel features of our interaction description language affect the level of the “Abstract User Interface” [332]. At this level, the descriptions of “Abstract Interaction Objects” [149] are independent of the concrete platform and interaction modality and are, rather, described in terms of their “semantics” [170]. While Paternò and Meixner refer to the semantics of user interfaces, we consider the semantics of the interaction itself and identified three concrete components, or dimensions, of the interaction semantics that we detail in Section 3.3.2. With respect to the concrete interaction abstraction categories, our language is related to the Dialog and Interface Specification Language (DISL) [86] and to XForms [327]. DISL proposes eight basic widgets for user interaction (*variablefield*, *textfield*, etc.). XForms controls include abstractions such as *trigger* (activation of a process) and *secret* (entry of sensitive information in a form). We also propose a set of basic interaction abstractions, but do not mix information that relates to the type of data exchanged with the high-level semantics of an interaction (see Section 3.3). This abstraction and separation of concerns, combined with the bidirectional communication between interfaces and interactive components, is the key to our descriptions being easy to understand and produce and still being expressive enough to cover all our considered use cases.

3.8 Summary

Most user interface description languages model user interfaces as composites of interactors such as text input, value selection, and output widgets where the appropriate interactor for an interactive component is selected based on its data type. We instead propose a way to express the *semantics* of an interaction which enables the generation of more intuitive graphical widgets, but also the mapping of interactive components to

gesture-based, speech-based, or physical interfaces. One main advantage of our approach is that the provisioning of a live interaction mechanism is reduced to the embedding of simple interaction information into the representation of a smart thing. Decomposing devices into atomic components and adding a small amount of simple information to collections of resources has proven to be well-suited for describing resources in an expressive but still easy-to-use way. The high level of abstraction of this information allows for the generation of modality-independent user interfaces while taking into account the capabilities of the target device. Smart things themselves do not need to be aware of what types of devices (e.g., PCs, handheld devices, or even other smart things such as Web-enabled knobs or switches) use this information and what kinds of interfaces these provide for users to control them.

Based on this approach, we presented a taxonomy of typical high-level interaction semantics and a description scheme that allows for the automatic generation of intuitive user interfaces for smart physical things and software components. We described a mobile device controller that generates user interfaces for smart things that embed a description of their interaction semantics according to our proposed language. The evaluation of the prototype in a laboratory deployment as well as in several deployments in private homes produced good results in terms of the usability of the generated interfaces and the generality of the description language: the application can generate convenient user interfaces where the user can choose between graphical, haptic, and speech-based interfaces. Our taxonomy of interaction abstractions covers all devices (physical and virtual) that we tried to include in our deployments as well as via mock-ups such as a lecture theater control system. Finally, a study of 780 participants showed that our proposed concepts for a user interface definition language can be used by tech-savvy individuals without any special training.

CHAPTER 4

Object Recognition for Direct Interaction with Smart Things *

In the previous chapter, we focused on the direct *interaction* between humans and smart devices, and showed how information about an appropriate user interface for controlling a specific smart thing can be embedded in the Web representations of the individual components of that device. We also mentioned several methods of *selecting* smart devices that range from simple approaches – for instance, entering the URL of the device – to more advanced ones, such as scanning barcodes or passive NFC tags, or using information about the location of the device to interact with. In this chapter, we discuss the use of *computer vision techniques* – in particular, fiducial markers and visual object recognition methods – for recognizing and interacting with smart things. Exploring how these and other technologies can be used by people to select and interact with devices in smart environments is a topic that is core to the domain of *Physical Mobile Interaction (PMI)* [25].

4.1 Interacting with Smart Environments

PMI encompasses research related to the use of *Near Field Communication* (NFC), *Radio-Frequency Identification* (RFID), and *visual identification* by means of tags (barcodes or 2D tags) or using markerless object recognition methods for interacting with physical devices. Handheld devices (e.g., smartphones) and personal wearables (e.g., smartglasses) increasingly support these technologies, thus allowing the interaction with other physical objects and enabling applications such as mobile product identification, mobile payment, and electronic keys. Relying on similar technologies and sharing the vision of user-centricity, developments within the PMI domain are also closely related to the Internet of Things and are an important enabling factor for the digital augmentation of our lives. Since the early examples for linking everyday objects with digital resources using RFID tags were presented in 1999 [234], many approaches to mobile interaction with physical objects have been investigated: the techniques of touching (NFC), pointing (laser pointer), and scanning (Bluetooth) have been implemented and thoroughly com-

*This chapter is based on the following published articles: [141, 142]

pared in [202] in the context of smart home applications, and the benefits of interacting with RFID-tagged paper maps using handheld devices were investigated in [187]. To enable the intuitive and direct interaction with displays, “Touch & Interact” [83] uses a grid of NFC/RFID tags that is attached to a screen and NFC-enabled mobile phones.

Already in the year 2007, researchers in the PMI domain considered mobile phones (i.e., today’s “feature phones”) and personal digital assistants (PDAs) to be suitable user interfaces for querying and controlling devices in smart environments because of their wireless connectivity, RFID/NFC tag detection capability, sufficient computational resources, and programmable screen and keyboard [194]: mobile devices have the potential to provide additional information about the state of an appliance that is not readily visible on its traditional interface and, to extend and personalize the user interface of the device. However, in the year 2007, a user study among 23 participants [194] showed that users are indeed faster when interacting with a device using its *traditional* user interface rather than an interface implemented on a mobile phone for *everyday tasks* (i.e., tasks that are most typical for a device and are performed very frequently, such as brewing a coffee on a coffee machine). According to [194], mobile user interfaces thus merely offer greater value to users than the traditional, attached interface of a device for *problem solving tasks* (e.g., resolving malfunctions of the device), *control tasks* (i.e., adjustments of device settings), and – to a lesser extent – *repeated control tasks* (i.e., control tasks that the user is familiar with). However, also with respect to these task types, the authors conclude that the added value of a mobile interface should not be attributed to the capabilities of the mobile device but that it “stems from the shortcomings of the physical user interfaces and the corresponding manuals.” The study also revealed that for 74% of the participants it was not an option to access all functions of the appliances only using the mobile interface.

We believe that it is time to revisit these observations, for several reasons: in the past few years, personal wearable computers such as smartglasses and smartwatches have appeared that set out precisely to increase the convenience and reduce the time needed for users to interact with smart devices in their environment [142] and with remote services [215], in particular to support users with everyday tasks. Furthermore, projects such as MARIA [171] and our own work on automatic user-interface generation (see Chapter 3) enable the creation of interfaces that support multiple interaction modalities and allow to take into account user preferences when generating an interface. Users might also be getting increasingly accustomed to using more dynamic interfaces than in the past – in a comparative study that was carried out in the year 2010 [84] all participants preferred the NFC-based interaction with dynamic displays to interacting with traditional screens.

Another reason that leads us to believe that the usage of mobile devices to interact with smart things for accomplishing everyday tasks should be investigated anew is that the above-mentioned study focuses on tasks that require interactions of comparable complexity when executed directly on the appliance and when using the mobile interaction device [194]. In our view, this takes away one of the main advantages that remote user interfaces can offer because one of the main strengths of virtual, software-based interfaces is that they can be reconfigured to match the tasks that a user requires most frequently. Additionally, they can be personalized for specific users on their own interaction devices,

and allow the creation of task-centric user interfaces that do are not necessarily dedicated to enabling interaction with a single device (see Section 3.2). Indeed, remote user interfaces are now commonly classified according to the functionality they provide *with respect to* the attached user interface of a device, as *complementary*, *duplicated*, or *detached* [116]. This emphasizes that the main idea behind the development of mobile interaction devices is *not to replace* traditional user interfaces, but to augment them with added functionality, tailor them to specific use cases, and personalize them for more efficient user interaction. This is exemplified by a study among user interface/experience designers described in [46]: participants chose more often to offer tasks only on the mobile interface (50%) than only on the physical interface (14%), and frequently made use of both (37%). When asked to explain their approach, the study participants referred to three factors that influenced their decisions: how *frequently* the task had to be performed, how *complex* it was, and whether the user needed to be *at the appliance* to accomplish the task.

4.2 Device Selection using Visual Object Recognition

Apart from providing users with intuitive interfaces to interact with smart things in their environment, for instance by using embedded interface descriptions (see Chapter 3), universal mobile user interface devices require a way of identifying the device a user wants to interact with. To resolve this “active device resolution problem” [97], one technology seems to stand out as an ideal candidate, due to recent advancements in that domain: *visual object recognition*. Theoretically, if it were possible to identify smart things in the user’s surroundings with high accuracy using only a camera – a standard component of smartphones and many wearable personal devices such as smartglasses – we would be able to make device selection simple for end users, in a non-intrusive way: users would merely need to look at a smart thing to interact with while wearing their smartglasses and the system could identify that device and load an appropriate interface for the user (see Fig. 4.1). The advantages of this approach are manifold: using a camera to select devices represents a straightforward and easy to understand mechanism for end users. In particular, smartglasses make it possible to know exactly what the user is looking at and do so in a non-intrusive way, meaning that the application could run in the background and provide “always-on device interaction assistance” to the user.

The underlying computer vision technologies that are required to accomplish this goal are, however, non-trivial to implement and tune to the task at hand, and error-prone: while the usage of visual object recognition methods therefore seems to represent an ideal solution to the active device resolution problem, it is, at the moment, limited to scenarios where devices have enough distinctive features to allow the software to differentiate between them. Furthermore, they depend on environmental conditions, such as enough lighting.

After a discussion of several prominent methods for visual object recognition in Section 4.3, we present the results of a study of the performance of these methods when applied in the context of the device selection in smart environments in Section 4.4. Afterwards, in Section 4.5, we present an extension of our generic mobile user interface

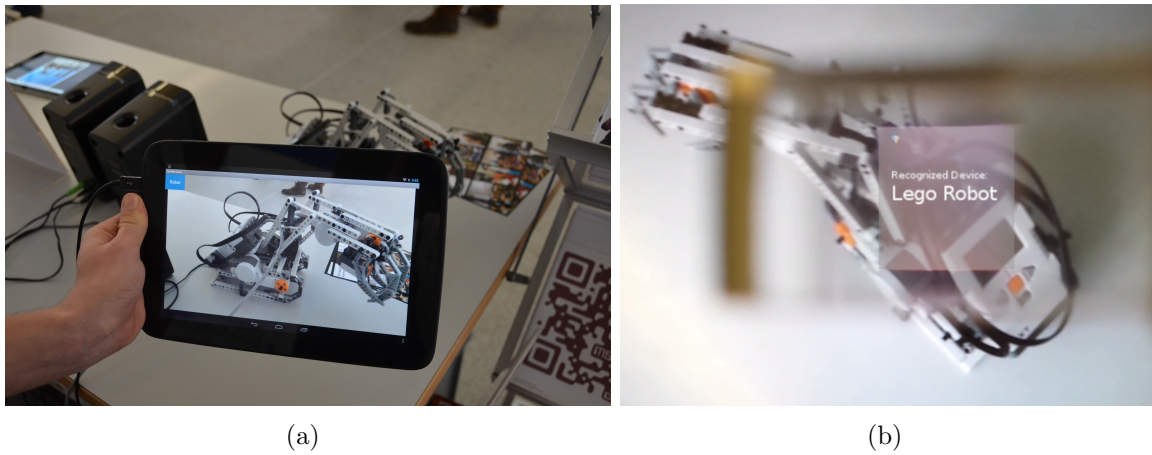


Figure 4.1: Selecting and interacting with a smart thing using visual object recognition on a handheld device (a) and on smartglasses (b).

application (see Section 3.5) with computer vision technologies and briefly discuss the results of a study of its usability for end users. We also present a system that combines visual object recognition on smartglasses with automatically generated user interfaces on smartwatches which, in our opinion, represents an ideal combination of the respective strengths of these two types of wearable devices.

4.3 Foundations of Visual Object Recognition

The domain of visual object recognition is a highly researched area that receives much attention from both, academia and industry. In particular, we have witnessed a surge of new startup companies in the image recognition domain over the last few years who aim to exploit this technology for marketing and advertising purposes and augmented reality applications – for instance, recognizing devices in supermarket shelves could revolutionize the way people shop, as detailed information about product properties such as their carbon footprint, allergy warnings, and competitive offers would be readily available [299]. One example of an application that can recognize everyday items such as ketchup bottles and DVD covers is Blippar [252]. Given an object with a distinctive pattern such as a logo or distinctive design and without occlusions, Blippar achieves impressive results on ordinary handheld devices such as smartphones.

Several approaches have been proposed in the literature to visually recognize objects in an image – these are based on a variety of metrics such as similar general image properties (dominant color, etc.), appearance/pixel values, geometric structure, part-whole relationships, or enough similar descriptors of distinctive image features such as corners or edges [253]. The general idea of systems that are based on matching feature descriptors is to first detect common features in training images of an object (see Fig. 4.2(a)). A representation of the complete object is then computed over these descriptors, where the individual descriptors can also be quantized, for instance using the closest “visual words” [41] (see Fig. 4.2(b) for an example visual vocabulary). Once a database of such

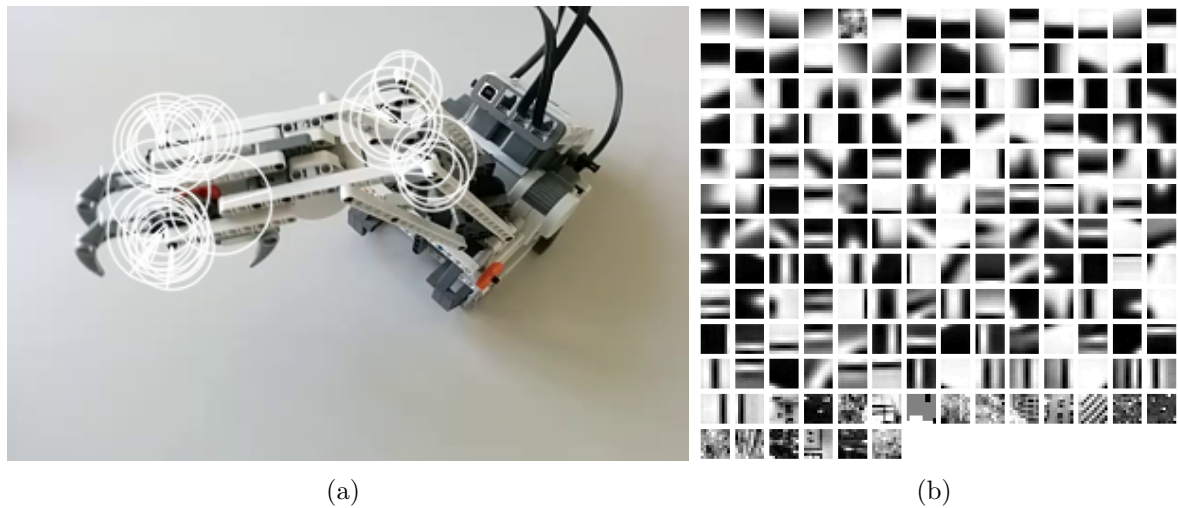


Figure 4.2: (a) Visualization of feature points on a picture of a device. Each circle represents a detected feature of the image and also indicates characteristics of its descriptor, in this case the dominant direction of features (the lines inside circles) and the scale at which the feature point was detected (the sizes of the circles). (b) Example visual vocabulary (from [115]).

representations has been created for all objects that are relevant to an application, the representation of a new image can be compared with all trained representations to find the object that best matches the image contents. Although this procedure sounds like a simple and straightforward categorization task, one must bear in mind that there exist about 10000 to 30000 object categories [19], and thousands of different objects per category.

A project that targets the augmentation of physical things with graphical user interfaces that are displayed on a handheld device in much the same way as we propose to is presented in [87]. This system exemplifies the difficulties of using visual object recognition techniques for this task: while it allows to render sophisticated 3D scenes on top of physical objects, it depends on 3D models of the devices and requires them to be printed with distinctive textures. We, in contrast, aim to create a system that enables users to select devices using their original visual features without any modifications and that only requires a few snapshots of the device for training its classification algorithm. Generally, our approach follows this procedure: for each camera frame, our software first extracts distinctive image features using one of several techniques and computes descriptors for each of these features (see Section 4.3.1; we additionally discuss the results from a comparative evaluation of many feature detectors and descriptors in smart environments in Section 4.4). Next, the application clusters the features of an object and matches each cluster center to a codevector from our visual codebook (the codebook itself is constructed from the most representative features of training images). The image itself can now be represented by a histogram that counts the number of occurrences of each of these codevectors – this is compared to histograms that have been obtained by applying the same procedure to a number of training images, typically between 10 and 20 for each object. Finally, to find out which object is contained in a given image, the application feeds the codevectors histogram into several binary Support Vector Machines (SVMs; each of these

SVMs is trained to give a binary response with respect to a single object from the training set) and chooses the object that corresponds to the SVM with the highest response value.

In the following, we describe this procedure in more detail – note that the methods that are discussed here are not only used for object recognition but also in other domains such as scene modeling and tracking, robot localization and mapping, and panorama stitching – for instance, the AutoStitch algorithm [27] was one of the first applications of SIFT features (see below).

4.3.1 Feature Detection and Description

In computer vision, the concept of *feature detection* refers to the extraction of interesting “keypoints” of an image. Extracted features represent a more abstract representation of the information contained in an image – thus, they can be obtained from training data and then used to classify new images and the objects contained therein. Most approaches to feature extraction are based on detecting high-contrast regions of the image such as edges and corners, which remain detectable under illumination changes and other noise. However, in their most basic incarnations, these often have deficiencies: for instance, simple template-based matching is often sensitive to changes in image scale and thus not suitable for matching images of different sizes – depending on its curvature, a corner might be classified as an edge on a different scale, and vice versa. In time, however, advanced techniques for feature detection and description have been developed that are not only robust to scaling, but also to rotations and general affine transformations of the image – this in principle enables robust classification even under larger viewpoint changes. After detection, features have to be described and stored in a way that preserves their distinctiveness and at the same time allows for fast matching. The idea behind such feature descriptors is that these shall capture the main visual characteristics of the local image region that surrounds a feature – processing the same feature as seen from different viewpoints and under different lighting conditions should yield similar descriptors, which is why invariance to affine transformation and lighting is important [42].

In the following, we review several methods for feature detection and description before showing a comparative evaluation on a training set with objects relevant in the context of pervasive computing. In particular, we emphasize the features’ invariance characteristics with respect to different image transformations and aim at capturing the essence of the developments that were achieved in this field over the last 15 years. One general trend that is visible in the domain of computer vision algorithms is that they are increasingly tuned to run on mobile devices with lower computing power and memory – for this reason, the last few years have seen many techniques that use *binary features* which allows for fast comparison and matching [3].

SIFT. Perhaps the most well-known feature detector and descriptor is the *Scale Invariant Feature Transform (SIFT)* [118]. In essence, SIFT makes use of standard operators (i.e., edge and corner detectors) to detect features and then uses local information to refine their descriptions: after detection, SIFT computes for each keypoint a distinctive descriptor vector from the keypoint’s surrounding region. This vector contains 16 gradi-

ent orientation histograms each of which encompasses 4x4 pixels from a 16x16 pixel local neighborhood of the feature. Because each of the orientation histograms has 8 bins (one bin per directional unit vector), the SIFT descriptor vector has 128 dimensions. Once computed, the vector is normalized and a threshold is applied for better invariance to illumination changes. The SIFT representation of a feature is invariant to uniform scaling of an image and to changes in orientation. SIFT can partially cope with local affine distortions (i.e., it exhibits limited viewpoint-invariance).

SURF. The *Speeded-Up Robust Features (SURF)* algorithm [12] is a scale- and rotation-invariant feature detector and descriptor that achieves much higher performance than SIFT with often only a small shortfall regarding the repeatability and distinctiveness of the detected feature points. Its high performance stems from SURF using integral images and describing the orientation of a keypoint by approximating given block patterns rather than explicitly computing gradient histograms [201]. Since they contain only 64 dimensions, SURF feature vectors are also considerably smaller than those of SIFT.

FAST. The *Features from Accelerated Segment Test (FAST)* algorithm [200] detects keypoints by considering, for each pixel, a circle of other pixels around that keypoint candidate and measuring the difference in saturation between the candidate and each pixel on that circle. If a sufficient number of pixels on the circle are brighter than the center pixel, that pixel is classified as a corner – for instance, the 9-16 mask of FAST requires 9 consecutive pixels in a 16-pixel circle to meet this criterion. The primary goal when creating FAST was to construct a keypoint detector that can be used in real-time applications at frame rate [200]. Indeed, in the tests shown in [201], FAST requires below 7% of the available inter-frame processing time when applied to a live video stream while other detectors are not able to operate at frame rate in this setting (for instance, SIFT requires 300% of the available processing time). For this reason, FAST is widely considered one of the best methods for keypoint detection in real-time scenarios [201] – in particular for use cases that require simultaneous localization and mapping [103]. However, FAST must be augmented using pyramid schemes to enable it to detect keypoints at different scaling levels and it does not include an orientation operator [201].

ORB. *Oriented FAST and Rotated BRIEF (ORB)* features [201] represent an amalgam of modified FAST keypoint detection and the *Binary Robust Independent Elementary Features (BRIEF)* [30] descriptor. It has been demonstrated that ORB is multiple orders of magnitude faster than SIFT but often exhibits similar precision in object detection, which is to a great part due to BRIEF being a binary descriptor and thus allowing for matching based on the Hamming distance between two vectors (instead of the Euclidean distance that is used for SIFT and SURF) [201]. For computing the orientation of FAST features, the authors use the intensity centroid of each corner feature, a simple yet effective method of estimating the dominant orientation of a feature [199]. After detecting such oriented FAST keypoints, ORB consequently makes use of modified, “rotation-aware” BRIEF feature descriptors (BRIEF, by itself, is very sensitive to image rotation and scale

changes [113]). As suggested by [201], our implementation furthermore uses a pyramid scheme to detect keypoints at multiple scales.

BRISK. The *Binary Robust Invariant Scalable Keypoints (BRISK)* algorithm [113] is a method for keypoint detection and description that, similar to ORB, is based on FAST for keypoint detection. It also achieves scale-invariance by sampling at multiple scales using a pyramid scheme. However, to determine and describe the orientation of a keypoint (and, thus, achieve robustness to rotation), BRISK samples the neighborhood of a keypoint in a distinctive pattern of locations on concentric circles around the keypoint. From a subset of the intensity values at these sampling points that only includes points that are sufficiently spaced apart, the overall pattern direction of the keypoint is computed and used for its description. Although BRISK achieves a matching performance that is only slightly worse than that of SIFT and SURF, it was shown to be much faster to compute, with a detection and extraction time per keypoint of only 0.037 ms compared to 0.428 ms for SURF and 6.156 ms for SIFT [113].

FREAK. The *Fast Retina Keypoint (FREAK)* descriptor [3] is based on BRISK but uses a sampling pattern for orientation-estimation that mimics the distribution of ganglion cells in the human retina. Because this pattern is less dense than the BRISK pattern, FREAK descriptors can be computed still faster while delivering equivalent or superior results than BRISK, SURF, or SIFT with respect to the number of correct keypoint matches under rotation, scaling, viewpoint changes, brightness shifts, and Gaussian blur [3].

Other feature detection techniques. In this overview of feature detection and description techniques, we did not cover methods that are computationally too expensive for our use cases, such as approaches that are based on dense features (e.g., DAISY [223]) and techniques that we discarded after a preliminary study (e.g., STAR [1] or MSER [123]).

4.3.2 Visual Vocabulary and Quantization

After image features have been extracted and described using one of the methods outlined above, they are aggregated to yield a representation of the image contents as a whole. In our work, we use the *Bag of Words (BoW)* model [41] to accomplish this. BoW has its roots in texture recognition (textures can be represented as histograms of the frequencies of their basic repetitive elements) and in word frequency counting as applied in the text mining domain (e.g., to generate orderless representations of documents such as word clouds) [300]. To apply this technique when describing arbitrary images, the image features are first clustered and each cluster center becomes a codevector from a codebook, or *visual vocabulary*. This process is tightly coupled with quantization, as the size of the codebook can be varied and feature vectors are mapped to the nearest codevector in the codebook. After this process, each image can be represented as a histogram that contains the relative frequencies of the occurrences of the individual “visual words” in the image, i.e., a histogram over the codewords from the visual vocabulary.

4.3.3 Image Classification

The final step of our object recognition pipeline is the classification step, where the descriptive image vectors that have been obtained either directly from the individual features or using approaches such as the BoW model are compared. For the comparison, our method of choice are binary linear SVMs that attempt to identify the hyperplane that maximizes the margin between the positive and negative examples for the occurrence of an object in an image [300]. As recommended in the literature [51], our classification step combines multiple binary SVMs – one per trained object – to reach a decision about which object is present in an image. In particular, we adopt a “one vs. others” approach, meaning that we train an SVM for each object by treating all images from our training set that do not contain this object as negative examples. In the classification, we then query all SVMs and take the result of that machine with the highest decision value. Alternatively, in a “one vs. one” approach, an SVM would be created for each pair of classes and the classification phase would involve a vote from each of these SVMs.

4.4 Comparison of Feature Detectors and Descriptors

The above review of different feature detection and description techniques already hints at the main trade-off that is involved when selecting an “optimal” object recognition algorithm: high-quality algorithms such as SIFT are invariant to a number of transformations but computationally too expensive whereas feature detectors and descriptors that can work in real time (such as FAST) suffer in terms of reliability and robustness [113]. A number of techniques (e.g., ORB, BRISK, and FREAK) aim to fill the void, to varying degrees of success in different scenarios. However, “choosing good features usually [is] the hardest part” [253] and, consequently, a number of extensive evaluations of feature detectors and descriptors exist in the literature (e.g., [113, 156]).

For our use cases, it is important to use a method that is sufficiently fast to compute on wearable devices with limited processing power and memory: for achieving high usability of the application, we want to be able to compute the feature descriptors fast enough for interactive usage – it should thus be able to classify at least 3 frames per second (FPS) on these platforms (higher performance is required for multi-object recognition, see Chapter 5). Our aim is to take into account as many features as possible within this time, but not to over-fit the trained model: features should not include irrelevant characteristics of the training images that are prone to changes but rather “high-quality” features that are suited for distinguishing objects. Finally, we want to find out to what extent the training can be delegated to end users themselves and, consequently, elicit the reactions of our system when training images are used that are not taken under highly controlled conditions.

We informed our decision on which of the available techniques to use via a comparison of several feature detection and description techniques in pervasive computing scenarios. Our evaluation comprises 16 different objects and is based on three sets of images per object at a resolution of 320x240 pixels. For training, we use up to 58 images per object from different perspectives and with a plain background (see Fig. 4.3(a)) – note that, to

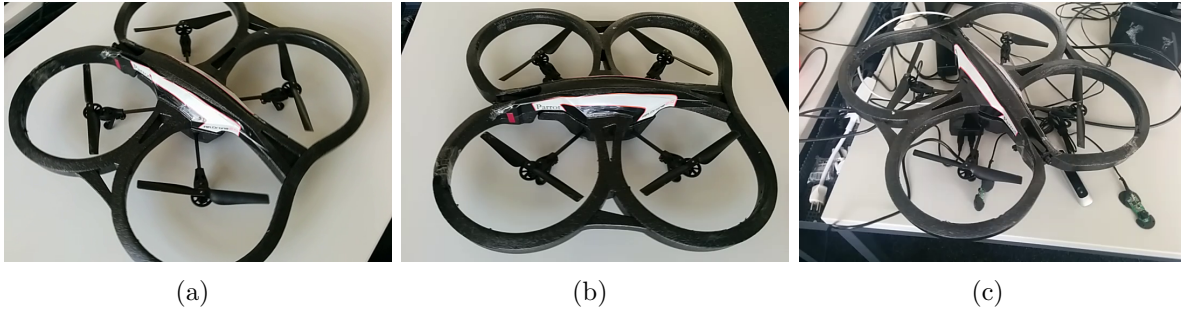


Figure 4.3: Example images from our training set (a), *Plain* test set (b), and *Challenge* test set (c).

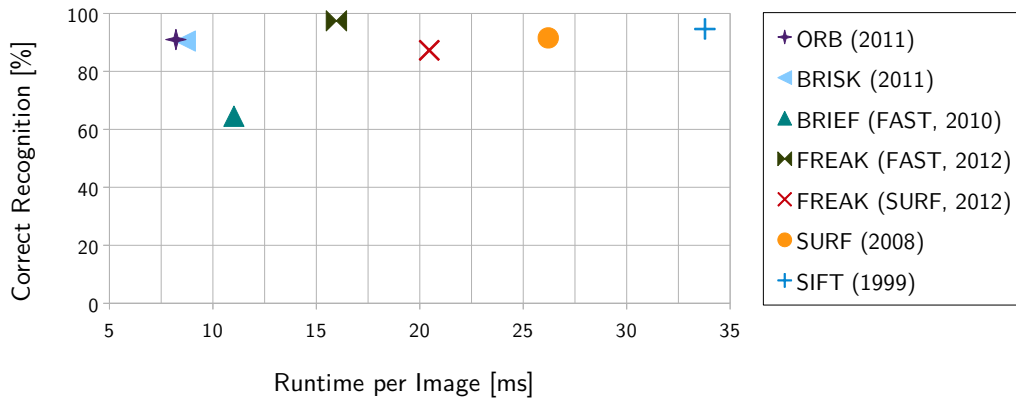


Figure 4.4: Comparison of the performance of several feature detectors and descriptors on our *Plain* test set.

simulate how users would register new objects in our system, the training images were intentionally not taken under laboratory conditions. For testing, we use two sets: the first, which we refer to as *Plain* was taken under the same conditions as the training images and consists of 190 images per object. This training set contains additional perspectives of the objects (see Fig. 4.3(b)) and we use it to find out how well the recognition methods perform under close-to-ideal conditions. The second, *Challenge*, comprises 190 images per object that were taken in environments with a lot of background clutter (see Fig. 4.3(c)) – these also contain perspectives of the objects that differ strongly from those present in the training set and that were taken under varying lighting conditions. We use the *Challenge* set to make statements about scenarios where end users train and use an object recognition system themselves, without any supervision. All training and test images were obtained by recording the scenes for up to 15 seconds and then splitting these videos to obtain the individual images – we consider this method an ideal approach also for end users who wish to extend the system’s capabilities by teaching it to recognize new objects.

We used the OpenCV framework [317] to evaluate the different feature detectors and descriptors – for this study, we used the default settings in OpenCV for each of the methods. The results of our evaluation on the *Plain* set are shown in Fig. 4.4: The ORB and BRISK detectors achieve a precision of over 90% while being the fastest in the set of eval-

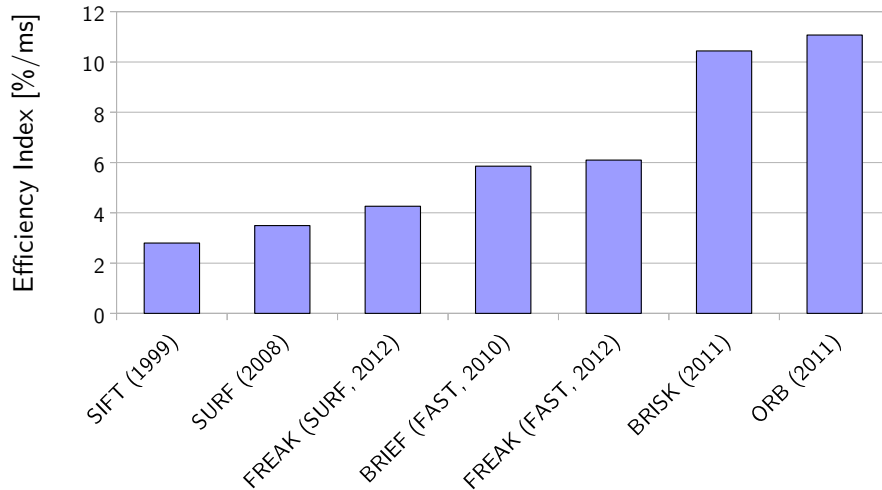


Figure 4.5: Comparison of all object recognition algorithms in our evaluation using a custom efficiency index that measures percentage-points per millisecond required for classification.

uated techniques (8.21 and 8.66 ms per test image on average).¹ The FREAK descriptor (in combination with a FAST detector) outperforms all other methods with respect to the precision of its object recognition (97.43%) but requires about twice the time needed by BRISK and ORB to classify an image. Considering the given dates of publication of the different methods, one can see the large performance gain that resulted from the conception of binary descriptors (SURF $\rightarrow$ BRIEF) and the iterative improvement of binary descriptors since 2010 (BRIEF $\rightarrow$ BRISK/ORB $\rightarrow$ FREAK). Note that the inferior performance of the BRIEF descriptor is mainly due to its high sensitivity to rotation, a drawback that is explicitly targeted by ORB. Because we strive for high precision as well as good performance, we additionally used *percentage-points per millisecond of time required* (per image) as an efficiency metric to rank the evaluated object recognition techniques: Fig. 4.5 shows that the ORB and BRISK descriptors/detectors are superior to the other methods with respect to this metric.

In response to our findings, we removed the FREAK (SURF) method that is clearly dominated by FREAK (FAST) and re-evaluated the performance of all techniques on the *Challenge* data set (see Fig. 4.6). In this setting, ORB keeps delivering good results at very high efficiency (57.86% at 10.06 ms per image) while higher precision can be obtained when using FREAK descriptors with FAST keypoint detection (63.42% at 24.01 ms per image).² Next, we removed all methods that are Pareto-dominated by others and optimized the remaining ORB and FREAK (FAST) methods further by modifying their input parameters and investigating their sensitivity to the number of training images.

Fig. 4.7 shows the performance of ORB and FREAK (FAST) on our *Challenge* test set in more detail. The algorithms clearly fail to reliably recognize the *WaterBottle*, *Kettle*, *LegoRobot*, and *LoudspeakerOffice* objects – for the kettle and loudspeaker, this

¹These evaluations were carried out on an Intel Core i7-3520M platform with 2.9 GHz and 8 GB of RAM.

²More detailed results for the performance of the FREAK (FAST) descriptor/detector with these settings including a confusion matrix with all objects can be found in Appendix B.

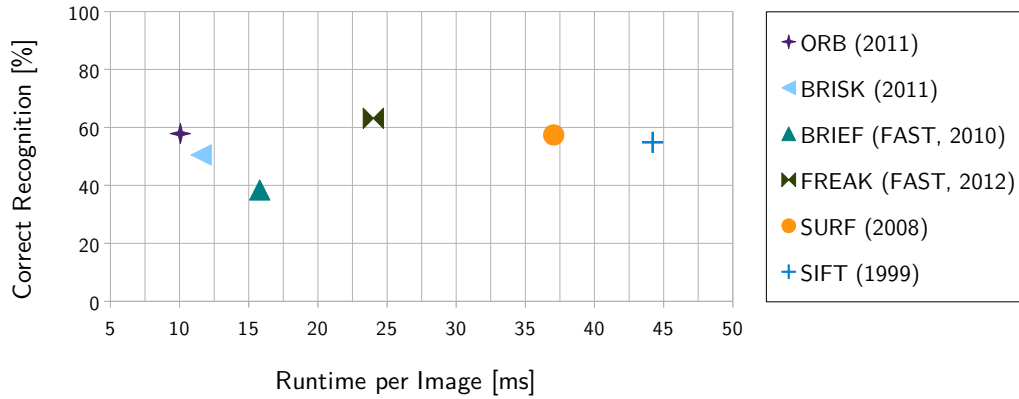


Figure 4.6: Comparison of the performance of several feature detectors and descriptors on our *Challenge* test set.

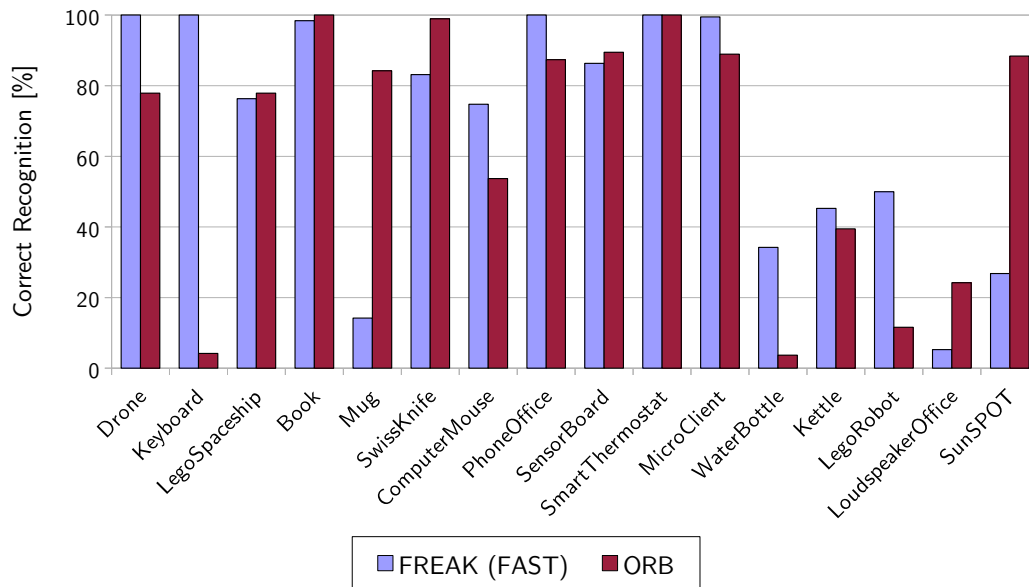


Figure 4.7: Performance of ORB and FREAK (FAST) on our *Challenge* test set.

is due to the test images having been taken from a wholly different perspective than the training images (see Fig. 4.8), suggesting that users should be instructed to try and capture training images from as many perspectives as possible. With respect to the water bottle and the toy robot, we attribute the poor performance of both methods to the very challenging set of test images (see Fig. 4.9). For other devices, however, such as the *Drone* (see Fig. 4.3(c)), especially the FREAK descriptors yield very encouraging results.

It is interesting to note that the two methods seem to complement each other on our test set (see Fig. 4.7; the correlation coefficient between their normalized recognition precision numbers is only 0.34), suggesting that a combination of both descriptors, while certainly being computationally more expensive, could lead to great improvements in the object recognition performance. Hypothetically, if we were able to select the better-suited algorithm for each individual object in our test set, such a fused approach would deliver a precision of 79.18% on average over all objects.



Figure 4.8: Example images from our training set (a) and *Challenge* test set (b).

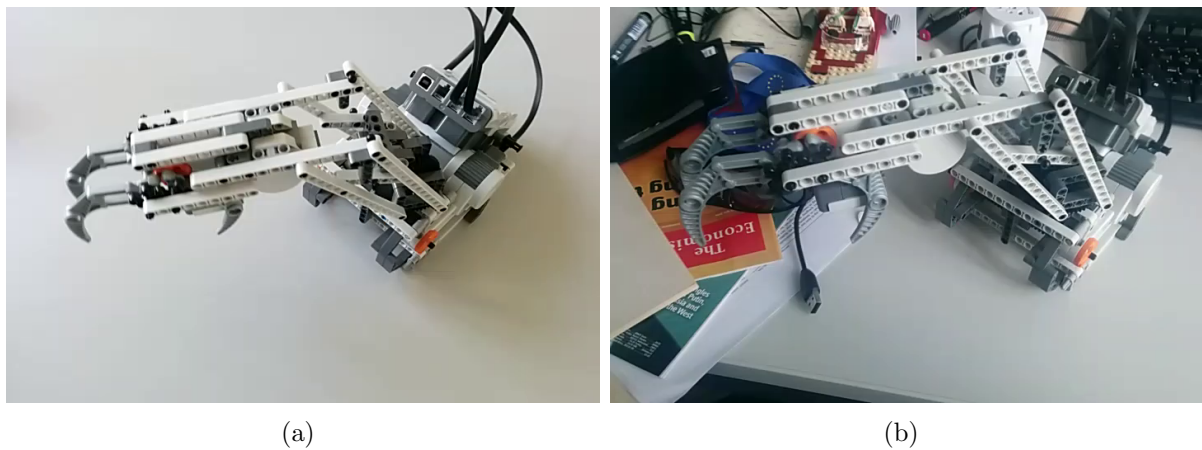


Figure 4.9: Example images from our training set (a) and *Challenge* test set (b).

Our evaluation of the algorithms' sensitivity to variations in the number of training images (see Fig. 4.10) shows that this is rather low and that around 20 training images are sufficient to achieve good results (68.39% and 64.38%, respectively). The algorithms still perform satisfactory (66.15% and 61.25%, respectively) when using only 10 training images, provided that these cover enough different perspectives of the object (as expected, due to the BoW quantization, the time required for classifying an image is not influenced by the number of training images used).

To improve the performance of the algorithms, we tested different settings for both: For the FREAK (FAST) approach, we varied the threshold values for the FAST detector and the pattern scale and number of octaves used for the FREAK descriptor. For ORB, we used different settings for the pyramid approach that ORB uses for scale invariance, for the number of features that are retained from a single image, and for the size of patches that are considered when constructing the descriptor. Finally, we varied the number of codevectors in the BoW codebook that is generated from the training images. In our tests, the influence of almost all of these parameters on the performance of the two algorithms was negligible. However, we were able to increase the precision of the ORB approach to 65.63% by making it invest a little more effort: we increased the size of the

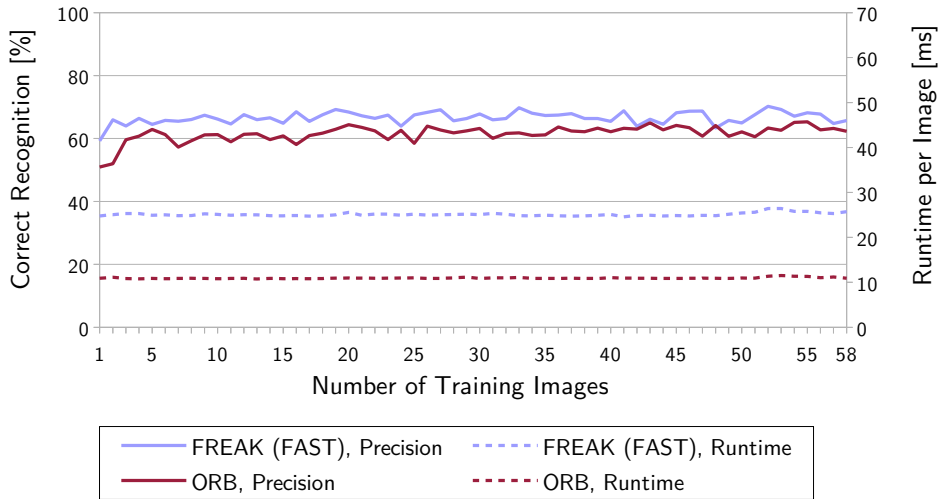


Figure 4.10: Influence of the number of training images on the performance of the ORB and FREAK (FAST) techniques on our sample set *Challenge*.

BoW codebook from 800 to 1600 entries and the number of features per image that the algorithm aims to extract from 100 to 200, which increases the time needed to classify a single image by about 60% to 16.24 ms.³ Increasing these values beyond 1600/200 did yield small precision gains (66.38% for 300 features per image and a BoW codebook size of 3200 codevectors) but at a steep cost: with these settings, the time required per image increased to 32 ms. When deployed on a Nexus 5 mobile phone,⁴ the time required by both techniques for classifying a single image increased by about a factor of 12, to 115 ms when considering 100 features per image and to 200 ms for 200 features per image for the ORB algorithm (these numbers were obtained at a codebook-size of 1600 codevectors). The FREAK (FAST) method requires about 322 ms to classify an image.

From our evaluation, we conclude that ORB and FREAK (FAST) descriptors represent good choices of visual object recognition algorithms for typical use cases in the pervasive computing domain. We expect their classification precision in real-world settings to settle in-between the performance numbers achieved on our two test sets, at about 70% to 80% – this, however, strongly depends on the concrete setting, especially the number and quality of object features, the number of trained perspectives, and the lighting conditions. For the deployment on mobile devices that have similar processing capabilities as current smartphones and on wearables with less computing power, we recommend using the ORB feature detector/descriptor for visual object recognition tasks: when deployed on a Nexus 5 smartphone, ORB can classify camera images at about 5 FPS, compared to about 3 FPS for FREAK (FAST).

³More detailed results for the performance of the ORB descriptor/detector with these settings including a confusion matrix with all objects can be found in Appendix B.

⁴The Nexus 5 phone has a 2.3 GHz quad-core Snapdragon 800 processor and 2 GB of RAM.

4.5 A Universal Remote Control for Smart Things

We used the process for recognizing objects using visual features described above to augment our universal user interface application discussed in Section 3.5 with advanced device selection capabilities. Using the basic version of that application, users had to scan an attached 2D barcode, use the NFC reader of their handheld device, or enter the URL of a smart thing to select it and start interacting – with the introduced modifications, they need merely point the camera of their user interface device at the smart thing to initiate an interaction. In the following, we discuss the integration of vision-based object recognition technologies with our interaction application for handheld devices in Section 4.5.1 and present the results from a brief study among users of the integrated application that aims at evaluating its usability for end users. Furthermore, we discuss an approach for interacting with smart things using a combination of personal wearable devices: in particular, the application that we describe in Section 4.5.2 uses smartglasses to recognize objects and then transmits an interface for the recognized device to the user’s smartwatch.

4.5.1 Intuitive Interaction with Smart Things on Handheld Devices

We extended our generic mobile user interface for smart things with an object classification algorithm to categorize items in its camera view. After recognizing a smart thing and thereby obtaining its URL, our application contacts the device to load a description of its user interface in the form of our user interface description language that was presented in Chapter 3. From this description, the handheld device renders a user interface and displays it as an overlay of the camera feed that is shown to the user (see Fig. 4.11(a)) while preserving the modality-independence of the interaction description (see Fig. 4.11(b)). The handheld device is thus transformed into a “magic lens” [20] that can render data provided by the smart thing and enables users to directly interact with devices using interaction primitives such as knobs or buttons (see Fig. 4.12), thereby extending and augmenting the attached user interface of the device, or replacing it altogether.

Typical use cases for such a system are abundant in the smart home domain, for

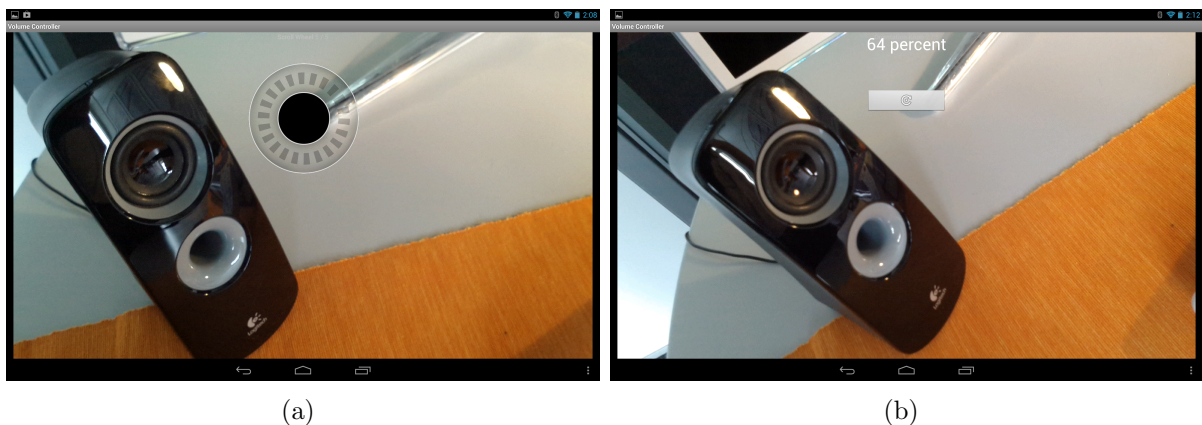


Figure 4.11: Interacting with a loudspeaker: (a) The application renders a clickwheel-like volume controller. (b) Users can also control the volume using a gyroscope-based interactor.

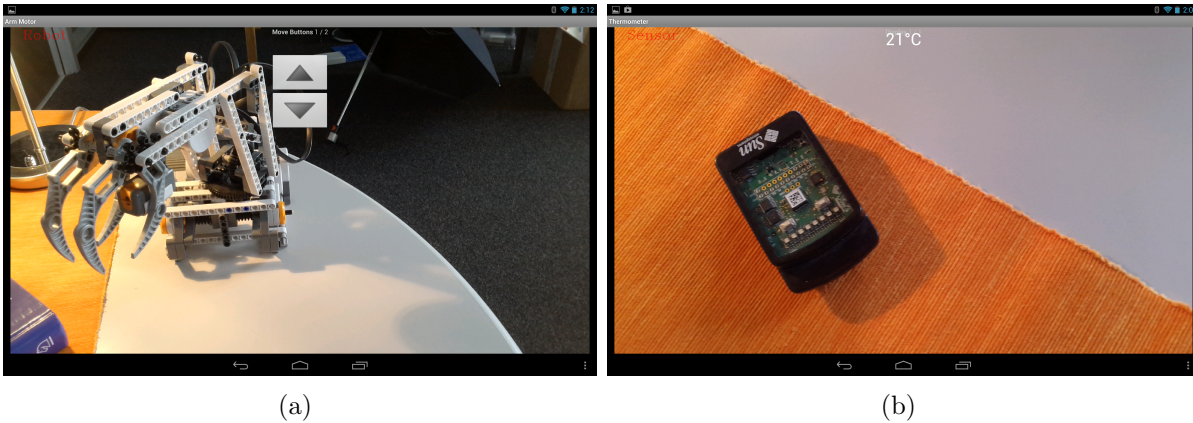


Figure 4.12: (a) Interacting with a toy robot: the application renders buttons to lower and lift the robot arm. (b) Interacting with a wireless sensor node: the application displays the currently sensed temperature.

instance to support user interaction with home automation systems, audio/video devices, smart thermostats, or toys. Additionally, we believe that our approach can be applied in industrial or medical environments, where already deployed devices could be augmented with a detached interface that gives operators more direct access to their sensors, and better control over their functions. Lastly, we discuss another use case for our interaction application in Chapter 8: interacting with Web-connected automobiles.

4.5.1.1 Visual Object Recognition

For recognizing smart things in the camera view of the user interface device, our software uses ORB features that are invariant to scale and lighting changes and exhibit limited invariance to viewpoint changes. We quantize the extracted features using a BoW model and use an individual SVM classifier for each object in our database to classify each camera frame. The classifiers are trained with 20 images per object from the training set described in Section 4.4. Depending on the quality of the training data, the visual features of the objects, and the amount of background clutter, we have shown this approach to be able to differentiate between up to 16 different devices: our application needs approximately 200 ms to process a single frame on a Nexus 5 smartphone, thus yielding an interactive frame rate of about 5 FPS (see Section 4.4 for details about the algorithms used and their performance) – however, because the recognition is done on a separate thread in the background, the implied lag is not noticeable by the user.

In our prototypes, the object recognition algorithms are implemented directly on the user interface device. A different option that we considered is to have the recognition step performed by a remote service, thus trading the challenge of processing on devices with limited capabilities for the issue of uploading frames fast enough to a Cloud service. We decided to implement our algorithms locally for three main reasons: First, our local classification algorithms already achieve satisfactory performance and sending images to a remote server for processing would yield higher response times [95]. Second, doing it remotely makes our system dependent on a persistent and high-quality Internet connection. Third, providers for object recognition services usually do not make their

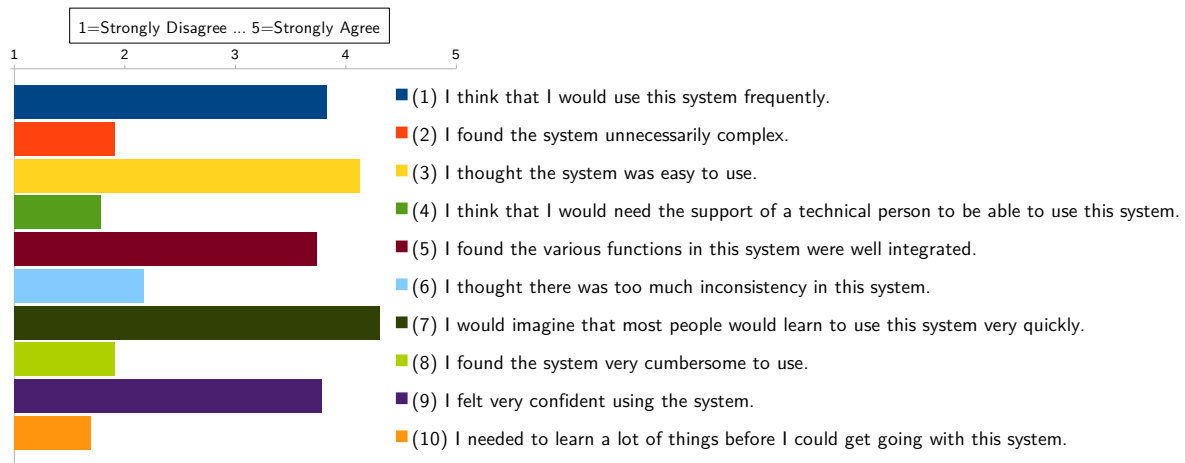


Figure 4.13: Results of a survey among 23 participants using the standard SUS questionnaire. Answers were given on a 5 point Likert scale (1=Strongly Disagree ... 5=Strongly Agree).

code open-source, meaning that our applications would depend on the benevolence of a company that might, at some point, decide to remove the functionality altogether.

4.5.1.2 Usability Evaluation

To get an impression of the usability of our prototype application, we conducted a survey among 23 participants – because of its small reach and because the study was conducted at a ubiquitous computing conference, its results can give an indication of our system’s usability but cannot be generalized. All participants used our system on a tablet computer to interact with a toy robot and a loudspeaker for about 1.5 minutes per participant. After using the tablet application, we asked them to evaluate the system using the ten standard questions from the System Usability Scale (SUS) [26], a popular questionnaire for subjective usability assessments [114]: our system achieved an average score of 75.76 on this scale. Although this number cannot be interpreted in a straightforward way, it shows that the usability of our system is higher than the average usability of systems tested using the SUS [114], and in the top-30% of tested systems according to [324]. The results of our survey are shown in greater detail in Fig. 4.13: in general, participants praised that the system was easy to learn and use (questions three, seven, and ten) but also mentioned that the system had too much inconsistency and that its functions were not well integrated (questions five, six and, to a lesser extent, question two).

During the survey, we used an early prototype of our application – we have since upgraded the feature detection and object classification from SURF features to ORB descriptors, as discussed above. The application can now also aggregate decisions for multiple consecutive frames to achieve more stable recognition results. Furthermore, we have made modifications to the interface of the application: for instance, in the version used during the study, the smart device to interact with had to be kept within the camera frame at all times during the interaction – as a consequence of the obtained SUS scores and feedback by users about this shortcoming, we changed the selection routine of the application to a two-step process: when the smart thing to interact with is recognized,

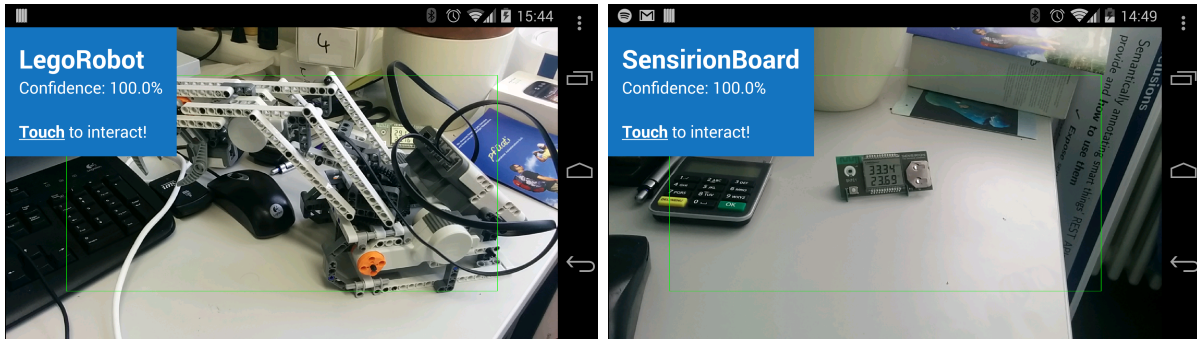


Figure 4.14: The second version of our universal smart things interaction application displays a button to initiate the interaction with recognized devices and shows the confidence of the classifier as well as an optional bounding box.



Figure 4.15: Our universal interaction application at ETH Unterwegs 2013/2014.

users are prompted to touch the screen to confirm their selection (see Fig. 4.14). The application then loads and displays an interface to interact with the device until the user explicitly ends the interaction with the object by again touching the screen. As a side-effect of this change, the application disables the object recognition thread while a user is interacting with a selected object, which results in a longer battery lifetime of the interface device. Furthermore, to increase the confidence of users when using the application, we now show a measure of how confident our system is about its decision and optionally display a bounding box to support users with aiming the camera at an object. An intermediary version of our application that already incorporated the changes to the user interface but not the novel classification algorithm was part of *ETH Unterwegs 2013/2014*, a traveling exhibition organized by ETH together with about a dozen Swiss secondary schools (see Fig. 4.15). During this exhibition, which aims to spark interest among students for ETH's degree courses in engineering and the natural sciences, the benefit of our changes to the application interface was confirmed.

4.5.2 Interacting with Smart Things using Personal Wearables

As already mentioned in the discussion of device selection techniques at the beginning of this chapter, a clear trend toward wearable computing is visible that is driven by the goal to reduce the time between a specific user intention and triggering the corresponding

action [215]. In an effort to combine the advantages of wearable devices with our approach of using visual object recognition methods for selecting devices, we present a system that combines personal head- and wrist-worn wearables to enable intuitive and efficient interaction with smart things for end users. Specifically, we propose to move the device selection task to the user's smartglasses while enabling users to interact with selected devices using their smartwatch: the smartglasses run a visual object recognition algorithm to identify smart things in the field of view of the user and, subsequently, an appropriate user interface is rendered on the smartwatch while taking into account a description of the interaction semantics of the target device (see Chapter 3). We refer to this separation of concerns between a head- and a wrist-worn device as “user interface beaming” – it enables users to discover and use interfaces of devices in their surroundings seamlessly and our approach could be applied for instance when interacting with appliances in smart homes, with medical devices in healthcare scenarios, and with devices in a smart factory.

In our opinion, the main advantage of smartglasses with respect to the interaction with smart devices in the user's surroundings is that they perceive the world from the user's perspective and can visualize information directly in front of users' eyes. However, the input capabilities of today's smartglasses are limited: Google Glass only provides a slim touchpad which severely limits its suitability as an interface for controlling a smart thing. Because using the built-in accelerometer of smartglasses that are available today as an additional input device is limited to few scenarios, the primary input of smartglasses is speech, an interaction mode that is often cumbersome to use, aggravated by shortcomings in speech recognition: especially when using speech commands in public, the recognition must be perfect to avoid annoyance and embarrassment. In contrast to smartglasses, the applicability of smartwatches to *select* devices using their camera is very limited – however, these enable convenient *interaction* with smart things due to the wrist-worn touch-enabled graphical user interface and advanced gesture recognition. Thus, as smartglasses seem to be ideally suited for device selection tasks and smartwatches provide rapid and convenient interaction capabilities, combining them and beaming user interfaces that are suitable for interacting with a target device to the smartwatch appears to be a prudent approach to achieve universal interaction of users with smart devices.

Our prototype system consists of smartglasses, a smartwatch, and a smartphone (see Fig. 4.16(a)). We use a Samsung *Galaxy Gear* smartwatch that provides a touch-sensitive 320x320 pixel resolution display and runs a custom Android on a single-core 800 MHz CPU. The watch can communicate with selected smartphone models via Bluetooth, but provides no direct Internet connection. For this reason, we use a Samsung *Galaxy S4* smartphone as communication hub to communicate with the interactive components of the target device on behalf of the smartwatch. We selected the *Google Glass* device as the head-worn component of our system. *Glass* features a dual-core 1.2 GHz CPU and a camera with a resolution of 720p.

For the object recognition, we ported the application that we described in Section 4.5.1 to *Glass* – this required only minor modifications to the application, as *Glass* also runs an Android operating system. Due to the limited memory and processing capabilities of that platform, our application runs considerably slower, but is still able to classify image

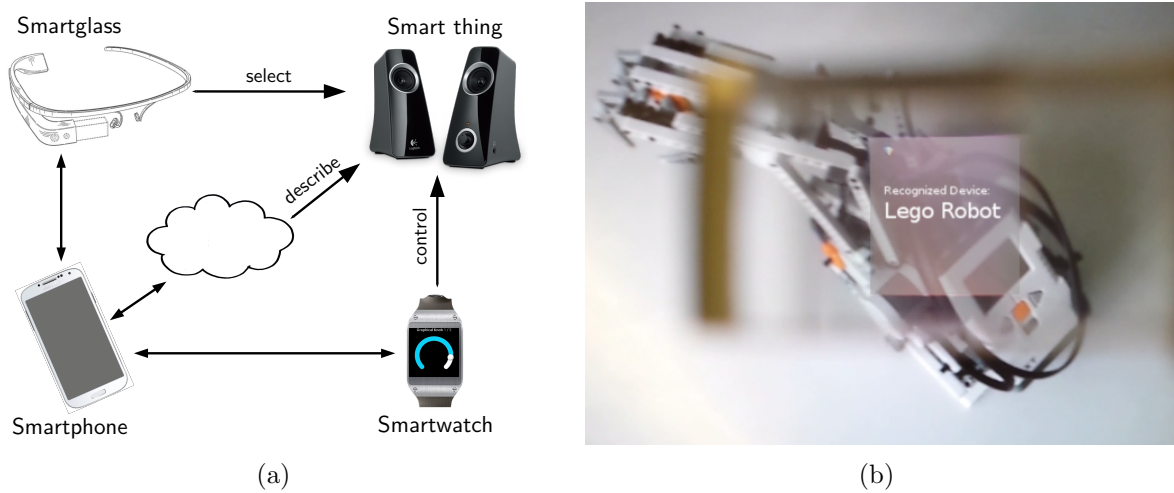


Figure 4.16: (a) System overview: the smartglasses recognize a smart thing based on its visual features; the smartphone downloads the user interface description from the Web representation of the identified device and beams it to the smartwatch; the user can control the object using the watch. (b) A recognized smart device seen through the display of Google Glass (images adapted from [142]).



Figure 4.17: Interfaces for controlling a stereo set on a smartwatch: (a) A graphical interface to control the volume. (b) An interface that switches the stereo set on/off upon shaking the smartwatch.

frames at a rate of about 0.5 FPS (see Fig. 4.16(b)). Whenever the device recognizes a smart thing, it can resolve its URL using a local database and send this information to the smartphone. The smartphone is responsible for fetching a description of a suitable user interface for the target device from the Web interface of the smart thing. It transmits the obtained interface description to the smartwatch that displays the described interface and allows the user to control the target using its touch screen and sensors (see Fig. 4.17).

We have implemented several example scenarios for our proposed system that include controlling the volume of an audio/video system, managing the temperature of a room by controlling the setpoint of a smart thermostat, and controlling a toy robot. For the A/V system, a graphical knob, gyroscope-based orientation knob, or virtual buttons can be used to control the volume of the system. Additionally, it can be switched on and off by shaking the smartwatch. The setpoint of the smart thermostat can be controlled using

interfaces similar to those for controlling the volume. Finally, the user can lower and lift the robot's arm using graphical buttons or an orientation switch that is triggered by the gyroscope of the smartwatch.

4.6 Summary

In this chapter, we discussed the challenge of supporting users when selecting smart things to interact with from a set of devices. We proposed to use visual object recognition techniques for this task, discussed several of the most common approaches in this domain, and showed an evaluation of how these methods performed in typical IoT settings. We furthermore presented an approach that enables users to interact with Web-enabled smart things on smartphones and tablets, as well as across boundaries of wearable personal devices, a technique we call “user interface beaming.” This system enables users to monitor and control devices using an amalgam of object recognition methods and user interface metadata provided by devices (see Chapter 3). We demonstrated that our system fully supports interface primitives that make use of different interaction modalities and that it is suitable for enabling convenient user interaction with different kinds of smart devices: in this chapter, we demonstrated its usage within the context of audio/visual equipment, smart toys, and devices common in smart homes such as thermostats and show that the proposed method can also be applied to facilitate user interaction with Web-connected vehicles in Chapter 8. We also envision the presented interaction system to be applicable beyond these domains, for instance in industrial and healthcare scenarios.

Our prototype applications on smartphones, smartglasses, and smartwatches represent initial steps toward seamless user interaction with smart things. However, in particular the capabilities of our object recognition software are limited – it supports only a predefined set of smart things and requires enough images of these for training the SVMs that we use for classification. To enable our prototype to operate within more dynamic settings in principle, we added the capability of automatically updating the SVM classifiers at runtime: when our application detects a significant change in the user's context (for instance, if the user moves to a different room), it can load classifiers for devices present in this new environment from a back-end server – and, consequently, unload classifiers for absent devices. This technique certainly does not eliminate the challenge of accurately classifying devices using their visual features – however, by dividing the problem of recognizing many different objects to a sub-problem with a smaller number of devices, it enables our proposed system to scale beyond enabling the interaction with only a handful of smart things.

In the future, we expect that it will be possible to further increase the accuracy of visual object recognition techniques used in systems such as ours – there has been significant progress in this domain since the conception of SIFT and SIFT-like detectors, descriptors, and quantization methods, especially in the last few years: ORB, BRISK, and FREAK, the detectors/descriptors that we determined to be most valuable for our deployments, have been proposed only in the years 2011 and 2012. Specifically, we believe that the integration of methods from beyond the visual object recognition domain can

yield significant progress for solving the active device resolution problem – for instance, expectations about which devices can be found in a typical living room, or a kitchen, can be used to constrain the set of objects from which the correct device must be selected, thus increasing the precision of the classification. Incorporating such ideas with traditional visual object recognition systems would allow computers to recognize devices in a way that comes ever closer to how humans achieve that task.⁵ In our opinion, the incorporation of information beyond the appearance of an object is necessary on the long run, as systems that only rely on visual information are inherently constrained. Rather, context information such as the user’s location, interaction history, and current activity as well as simple clues such as the current time should be taken into account when attempting to find out which device users wish to interact with in a given situation.

⁵The influence of expectations on the object recognition performance of humans has been demonstrated, for instance, in [28].

CHAPTER 5

Real-time Visualization of Device Interactions *

In the previous chapters, we discussed how Web technologies, embedded semantic information, and object recognition technologies can enable users to seamlessly interact with smart devices in their surroundings. The proposed technologies, however, restricted users to the *direct* interaction with *individual* devices – neither did we consider that these devices might be communicating with other smart things and services, nor did we allow for collaboration between devices in smart environments to better support the user.

In the rest of this thesis, we discuss these aspects of environments that are populated by Web-enabled smart devices: First, in this chapter, we show how the methods that we initially proposed for the interaction with smart things can be used to inform users about which services their devices are interacting with at any given moment, thus surfacing “device whispers” in smart environments. In Chapter 7, we propose a technique that enables devices to fuse their individual services for achieving higher-level functions by making them aware of what they are capable of doing by themselves and using semantic reasoning to automatically deduce service mashups. Because this approach requires a greater degree of control over the individual devices present in a specific environment and information about the services they provide, we precede this discussion with a proposal for a supporting management infrastructure for Web-enabled smart things in Chapter 6.

5.1 Eliciting Device Interactions in Smart Environments

We believe that keeping users informed about what their connected smart things are doing – and why – will become increasingly important as the Internet of Things penetrates our daily lives and many of our once isolated devices start interacting with each other and with remote services. In particular for the smart home domain, we expect that connected household appliances and entertainment devices will enable services such as remote monitoring and control, over-the-air firmware updates, and demand-side electricity load management. Thus, we expect that devices in future smart homes will not only interact with each other and with user interface devices as was stipulated in the previous chap-

*This chapter is based on the following published articles: [133, 137]

ters, but also with services that are hosted remotely by utility companies, manufacturers of household appliances, or firms that analyze a household's data to provide advanced services to its inhabitants.

The development of increasingly automated sensing and actuation technologies will enable powerful new services and applications that support smart home inhabitants. However, this development also gives rise to a number of challenges that, if not properly addressed, might hinder the widespread adoption of smart homes because people could lose trust in the smart things present in their homes and the remote services these devices use [185]: data that is produced by sensors in smart homes contains detailed information about the lives of its inhabitants that could be very valuable to companies. For instance, by analyzing only the aggregate electricity load curve of a household, it is possible to determine with high accuracy whether its inhabitants are employed, and to estimate the household income [14]. From the same data, the household occupancy pattern can be obtained and, potentially, it can be predicted when – and for how long – the home will be unoccupied on a specific date in the future. Having access to data of this kind could be of great value to insurance companies or providers of targeted advertising and we are duly witnessing increasing worldwide debate about the preservation of individual privacy in an increasingly connected world. Beyond the issue of sensitive data leaving the smart home, a “fear of pirating,” i.e., unauthorized control commands from outside that actuate devices in the home, constitutes the most severe anxiety of people regarding the usage of remote services by their devices: according to the authors of [192] who conducted a user study among smart home inhabitants, it is therefore imperative that “the user must always remain in control of the system” – this requirement includes that the user must be able to find out what services devices in his home communicate with and what the transmitted data is used for by service providers.

This is challenging because devices in smart environments form complex networked systems where communication takes place invisibly and “behind the back” of users and because devices potentially make decisions fully autonomously. Moreover, users already find it challenging to configure and maintain the rather simple device networks that are present in their homes today, mainly because of the “invisibility of settings and configuration information” as well as “poor strategies for diagnosis and troubleshooting” [182]. Managing a network requires inhabitants to use tools that are hard to use for them [72] and some expect that, as smart homes are growing increasingly complex, users will suffer a loss of control of their environment [185].

5.1.1 Network Management Tools for Commercial Installations and Home Networks

The above-mentioned challenges are neither new nor constrained to networks in smart homes: many commercial tools already exist that provide supervision and management support for computer networks. Examples for such tools are the *HP Network Management Center* that obtains traffic information by attaching a passive router as “Route Analytics Management System” and creates a real-time and historical record of routing.

Another example for such a system is the network management suite from SolarWinds, Inc. that contains a dashboard to monitor network traffic usage and can automatically detect anomalies. Other management tools include capabilities such as predicting future network statistics and are often extensible using plug-ins that can perform advanced analytics on collected traffic data. All the mentioned systems, as well as a plethora of other commercial products, however, focus on large networks in companies or university campuses and are usually designed for use by trained network administrators [72]. Consequently, they are not suited for usage by private users [182].

As the complexity of networks in private homes increased over the last decade, management tools have also been designed for this domain – these provide simpler user interfaces but typically are a lot less powerful than their counterparts in industrial installations. In particular, home network management systems often used to focus on static properties of networks such as parental controls or media prioritization. However, this seems to be changing with modern products such as the *LinkSys SMART Wi-Fi* software suite that provides the functionality to monitor which devices are using the network in real time. The *Private Eye* application lets users monitor which remote services their computer is communicating with – a concept that has already been demonstrated in the NetFlow project [153].

Many researchers today believe that the lack of adoption of tools for home network management such as those described above is due mainly to user interface issues, i.e. a failure of presenting the output of these tools in a way that is easy to understand for users [8] – this has been identified as the main reason for many home network users still relying on the indicator lights on their routers and cable modems to monitor and debug their networks [182]: granted, a tri-colored LED can only convey a very limited amount of status or activity information. However, it does so in a highly intuitive way.¹ Consequently, to bring the functionality of advanced network monitoring tools to everyday users, applications have been created that aim for higher usability by providing *graphical representations* of events that occur in the network. For instance, VISUAL [8], a tool that is part of the *Network Eye* security visualization architecture, enables administrators to understand traffic patterns and identify threats to the network by providing a graphical representation of incoming and outgoing connections. Indeed, many network management applications are targeted at intrusion detection, and the training of novice network analysts by visualizing what is considered “normal” network behavior – TNV [69] and [38] are examples of such tools, the latter can also be used via a mobile interface and shows detailed traffic data statistics. Finally, an interesting project outside traditional network management that, however, also allows for the visualization and analysis of network information such as association and disassociation events of mobile devices with access points is [172]. The authors demonstrate their system in the context of the analysis of crowd movements in a football stadium.

¹A very early project that visualized network activity very intuitively was presented by the artist Natalie Jeremijenko: the *Live Wire* system consists of a string attached to a stepper motor that twitches to dynamically indicate the amount of network traffic. Mark Weiser called the *Live Wire* prototype one of the first examples of *calm technology* [235].

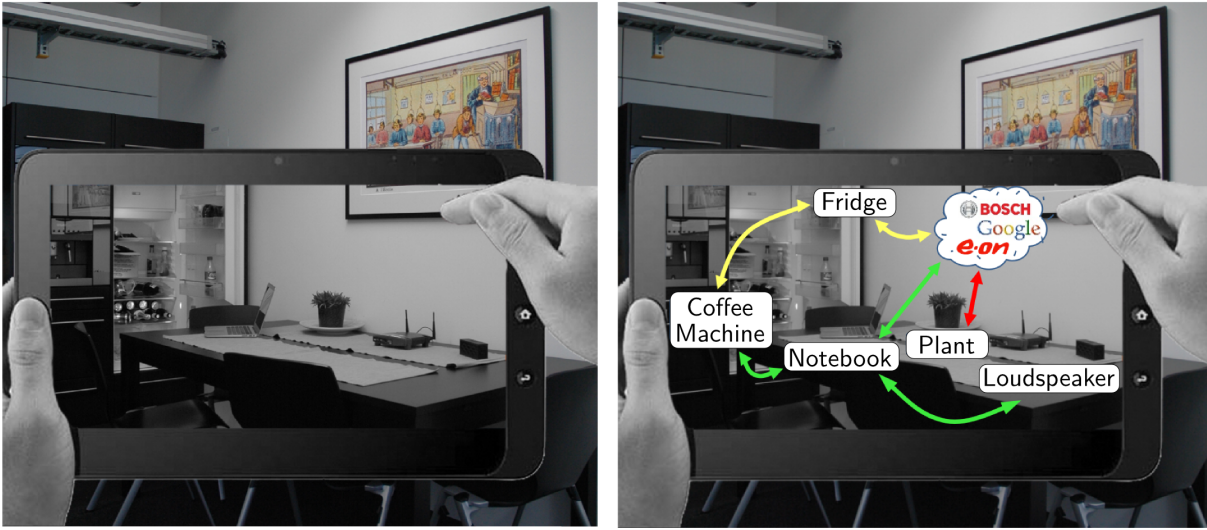


Figure 5.1: Device whispers in smart environments (mock-up; images adapted from [133]): We want to enable users to monitor traffic flows between devices using a handheld or wearable interface as a “magic lens” [20]. The application could also classify interactions as “safe” (green), “unusual” (yellow), or “critical” (red).

5.1.2 Moving Network Management into the Physical World

Recently, network monitoring tools have started to consider network nodes as embodied devices and applications such as CANVIS [154] were created that allow individuals to select networking devices using their smartphones to obtain information about how and how much they communicate with others. We propose to use visual object recognition technologies and methods known from the domain of augmented reality to take these approaches to the next level, by visualizing data about connections between devices and services on mobile devices as an overlay for the real, physical, world (see Fig. 5.1). This approach could allow users to easily and intuitively monitor information flows between nodes inside their home networks and with remote services and thereby get more in control of their network installations – similar ideas are applied for visualizing urban infrastructures, for instance in the Vidente project [203] where underground tubing is visualized using an augmented reality application on a smartphone. Our approach should enable users to better understand device associations in their smart home and to perceive unwanted interactions, where we aim to go beyond currently available network monitoring approaches also with respect to the inspection depth that our system allows: it is, using our methods, possible for users not only to see HTTP packets that are transmitted between devices, but also to inspect their contents in near real time. This has only rarely been done before, for instance in the *EtherPeg* tool [255] that can display a network graph and visualize image files as they are passed from one network node to another. Finally, our proposed network analysis software can not only be used by administrators to monitor information flows between devices, but also to control them using an approach that is known as *software-defined networking*: our system gives users the ability to control the flow of packets between devices in a smart home and remote services by configuring networking restrictions directly on their home networking hardware, by “cutting” visualized

connections on the user's handheld device.

In the rest of this chapter, we will present our network monitoring and control system in greater detail, starting with a discussion of how the necessary raw data about traffic flows in a network is obtained by our system. Next, in Section 5.3, we propose several methods of how this data can be presented to users: Our system can visualize the data in several different ways on traditional interfaces, for instance as an animated graph or on a timeline. It can furthermore track fiducial markers that are attached to devices to render device associations as a basic augmented reality overlay. Finally, it makes use of the object recognition techniques presented in Chapter 4 to associate devices to networking data about them and display this data as an augmented reality overlay on the camera stream of handheld user interface devices such as smartphones or tablets. In Section 5.4, we discuss several interesting services that are enabled by our system, in particular by its capability of inspecting HTTP packets and relating to software-defined networking, and propose multiple use cases for the proposed system.

5.2 Collecting Network Traffic Data

Before our application can visualize networking information, we must create a system that collects the necessary data in real time from the networking infrastructure or the networked smart things themselves. We implemented two different methods of collecting this data: First, we created an application that is deployed on devices that are part of the networking infrastructure (in our case, on a router) and passively “sniffs” packets that pass through this hardware. Second, we implemented several modifications to widely used software libraries for HTTP servers and clients that transparently add the functionality to log HTTP packets to applications that use them. We discuss both approaches in the following – the main advantage of the passive sniffing of packets is that the logger application can in principle record all packets that are exchanged in a computer network without modifications to the individual network nodes. However, low-level packet sniffing has several drawbacks: most prominently, it does not enable us to inspect message contents on the application layer, thus limiting the capabilities of our system with respect to a more in-depth analysis of the transmitted information.

5.2.1 Approach 1: Passive Sniffing

The first approach that we implemented to analyze interactions between Web-enabled devices and other services was a low-level packet sniffing application that is deployed on the network router (see Fig. 5.2). In our deployment, we used a Linksys WRT54GL router, a device that is widely used in home networks and that can easily be accessed and reconfigured because it runs the Linux-based OpenWRT operating system [316]. Our network sniffing application can thus in principle be deployed in private households as an add-on module to the router and without any changes to the networking infrastructure. The sniffer applications makes the collected data directly accessible for client devices via a Web interface.

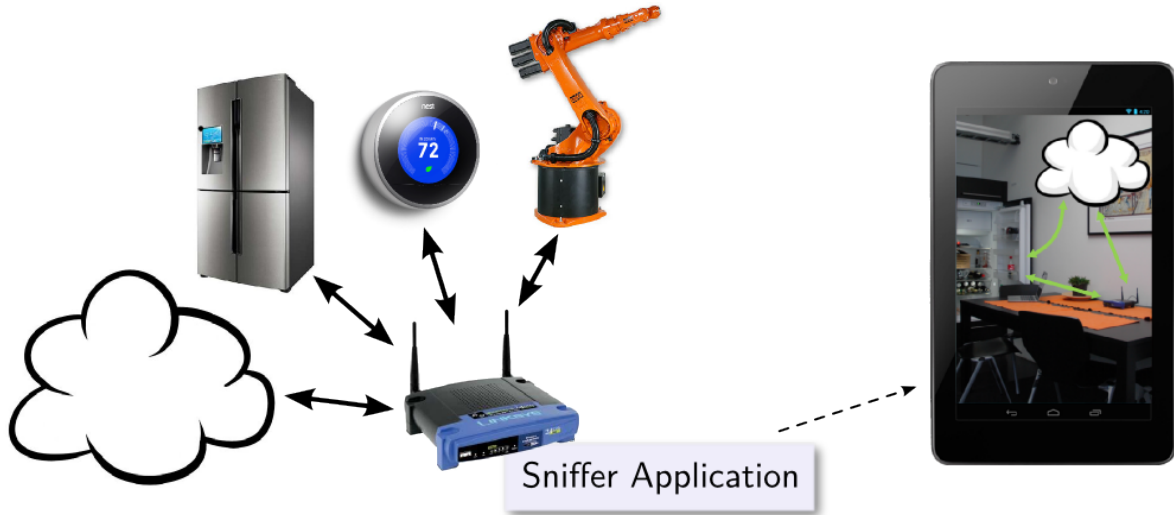


Figure 5.2: Our sniffer application is deployed on the network router. It records information about packets exchanged between network devices and makes this data available to user interface devices via a Web interface.

Sniffing packets with the help of the Linux *netfilter* firewall [308] is straightforward as it merely requires adding a new rule to the kernel firewall using the `iptables` command. We added a rule to log all packets that are not intended for local delivery, by modifying the `FORWARD` chain of the firewall (*chains* are the firewall’s abstraction for grouping together filtering rules). This rule causes information about all packets that pass through this chain to be written to a local system log by the *syslog* daemon – our application can then fetch the required information, process it, and store it to a local in-memory hash table that maps IP addresses of network nodes to information about their connections with other nodes. Already at this stage, our system was straining the resources of our hardware platform: we needed to use the *uthash* library [280] since *GLib* [276] proved too resource-hungry for our hardware. For the same reason, attempts to inspect network packets in more depth using the *packet capture* (*pcap*) API proved infeasible.

The router provides the collected data about device interactions, as well as information about these devices such as their name, a short description, and their URL, to clients via a REST API. By specifying the source and destination IP addresses of devices they are interested in within an HTTP request, clients can ask for information about interactions of these devices in one of three different modes:

- a) **Client defines source IP address:** The interface returns data for all IP addresses that the specified IP address has in the past established connections with.
- b) **Client defines destination IP address:** The interface returns data for all IP addresses that have in the past established connections with the specified IP address.
- c) **Client defines source and destination IP address:** The interface returns data for connections only between the two specified IP addresses.

For each request, the response includes information about the number of packets transmitted between two IP addresses as an account of the strength of their connection

and the protocols that were used in the communication (e.g., TCP, UDP, or ICMP).

An evaluation of the implemented system showed that our approach of relying on firewall rules to collect interaction metadata works in principle, but that the router hardware is not capable of handling heavy traffic, for instance during file downloads. Another drawback of the described system is that our hardware (and also many other network routers) behaves like a network bridge when handling traffic between wireless clients, to reduce packet processing overhead: consequently, packets that are transmitted from one wirelessly connected device to another are never passed to the kernel firewall, and thus never seen by our application.² While it is, thus, possible to collect device interaction data using networking hardware and without modifications to the communicating network devices, we showed that this poses multiple challenges due to the router's limited hardware resources. Since we, however, want to obtain not only information about packets passed between wireless endpoints but also more detailed information about the payload of network packets as well as the full source and destination URLs, we decided to adopt an active logging approach instead.

5.2.2 Approach 2: Active Logging

To overcome the disadvantages of the passive sniffing, we decided to deploy logging clients directly on the smart devices whose interactions we want to monitor and have these modules report metadata about communications between devices to a central storage back end (see Fig. 5.3). This component persists recorded messages and forwards the information in real time to visualization clients that are connected using HTML5 WebSockets [329]. Alternatively, user interface devices can query the REST interface of the back end to obtain information about past device interactions.

With this approach, we are not required to record communication metadata on the network level as for the passive sniffing, but can instead record interactions on the level that is most relevant for users in WoT scenarios: we directly log HTTP messages, including their headers, the full request URL, and the request and response payloads. However, as we now record this information in a distributed way, it is possible that the causal order of observed interactions at the back end is inconsistent with the global ordering of interactions between devices. Since one main purpose of our system is to help users monitor the causes and effects of device interactions in their smart environment, it is, however, important that the visualization of message flows between devices reflects the true causal ordering of these interactions. To enable this, we record the causal relations between interactions using the vector clock algorithm [125]: The vector clocks are piggybacked on HTTP messages between devices and our logging software takes care of merging the local and incoming vector clocks upon receiving a request or response.

To additionally capture interactions with external services and unobserved devices (i.e., endpoints that are not under our control), our system uses a combination of server-side and client-side logging, where the concrete setup is chosen dynamically based on which of the two endpoints in an interaction is directly observed by our software: If *both*

²A potential remedy to this problem is to use a mechanism called *ethernet bridge firewalling*. However, this technique is too resource-heavy for our hardware platform.

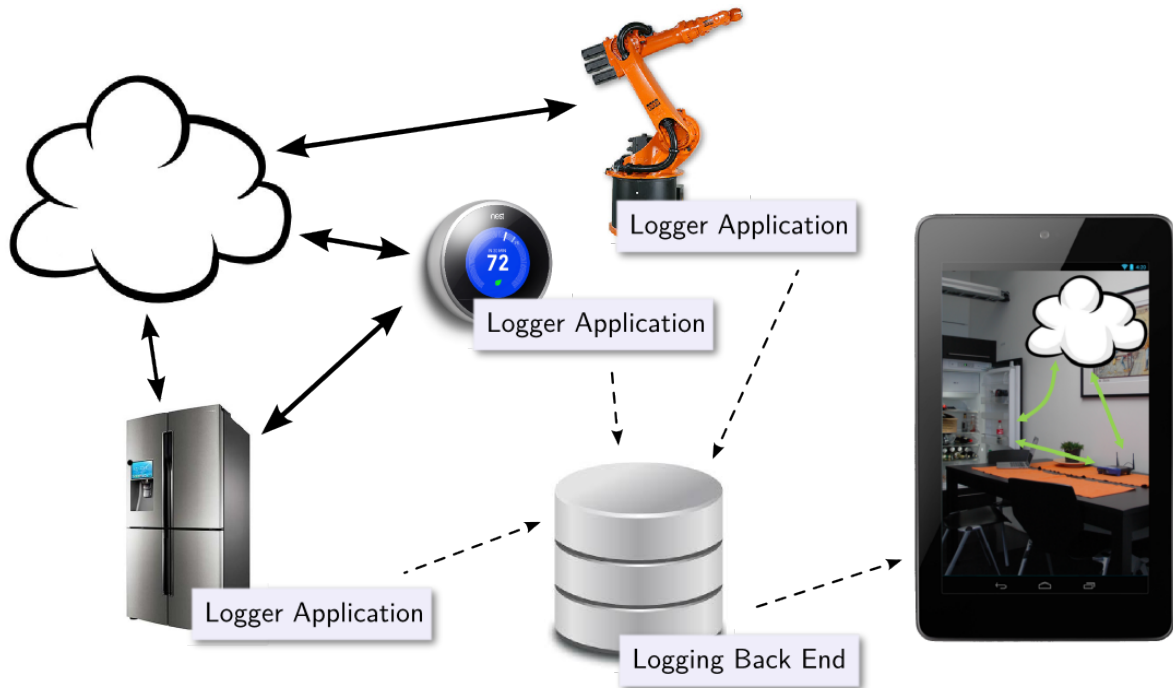


Figure 5.3: The logger application is deployed directly on network nodes. It records information about HTTP requests and responses between endpoints and sends this data to a central server. This server makes historical data available to clients via a REST interface and pushes real-time information about interactions to registered clients using HTML5 WebSockets.

the server and the client are observed, we record the HTTP request on the server side and the HTTP response on the client side. If *only the client* is observed, it takes care of logging requests and responses from external services and unobserved devices that it interacts with. Conversely, if any observed node *receives* a request from an unobserved client, it logs that request/response pair.

Monitoring Interaction Chains Since most Web interactions in a Web of Things context are part of a task that involves multiple HTTP requests and responses, our system does not only record isolated requests but aggregates them to *interaction chains*: all requests and responses within a chain are the consequence of a single initial HTTP request. By visualizing entire interaction chains, our tool thus enables users to inspect Web interactions within their context, allowing them to better decide whether, for instance, a specific request to an external service was intended and facilitating the monitoring of complex distributed interactions. To enable our system to visualize series of interactions between devices, each captured HTTP interaction must be assigned to an interaction chain. However, due to the stateless nature of HTTP we cannot understand the causal relationships between HTTP requests by merely observing network communication. Our logging software therefore adds an HTTP header entry that contains the identifier of the current interaction to each outgoing request. Additionally, whenever a logged Web server receives a request that contains an interaction chain identifier, it stores that ID and attaches it to each outgoing HTTP request that it generates while processing the incoming request. While this technique represents a sound heuristic of assigning requests

to interaction chains, it may happen that a server generates a new request *while* processing another (and, thus, assigns the two requests to the same interaction chain) although the requests have no causal relationship. Likewise, new requests that are generated *after* the processing of another request has finished are assigned to different interaction chains although they might be related to the previously received request. Assigning requests correctly to interaction chains in these scenarios, however, would require our logging system to reconstruct the entire logic of applications running on the smart devices, which is not the focus of our project.

5.2.3 Discussion of Logging Methods

In contrast to the non-intrusive logging presented in Section 5.2.1, the active logging approach requires that server applications report observed interactions to a central back end, attach the required HTTP headers to outgoing requests, and merge vector clocks of incoming responses. However, to avoid burdening developers of smart devices and other Web services with having to manually modify their implementations, our system uses *Java Agents* which can modify the bytecode of Java classes using so-called *class transformers*. Using this approach, the logging client can be deployed to Web servers that are based on the Grizzly NIO framework [319] and use `URLConnection` to make requests without the need of modifying a single line of code, by merely attaching our transformers when invoking the application (using the *javaagent* JVM parameter). We believe that this small change to the instruction that deploys applications on monitored smart things is worthwhile for several reasons and that the active logging approach should be preferred to passive sniffing: First, by logging requests at the application layer, the logging system is able to trace the execution path of a request and thus can reconstruct interaction chains from HTTP messages – information about which requests are related to one another is relevant for users when monitoring interactions between network devices. Second, we gain access to the full URI of an endpoint which allows to differentiate between different resources (e.g., sensors and actuators) that are served by a single server. In case the message is unencrypted, we also obtain its full payload which is valuable when attempting to automatically classify requests and warn users about interactions that seem suspicious – however, even when considering encrypted messages, the base functionality of our system, i.e., enabling users to monitor which devices and remote services their smart objects communicate with, is preserved. Finally, logging on the application layer is reasonable because it is not susceptible to routing optimizations that handle frames already on the link layer and where we consequently lose access to the source and destination IP addresses.

5.3 Visualizing Interactions

Both proposed approaches to collect information about network interactions involve a back end that can deliver recorded interaction metadata via a REST interface. In the active logging approach, visualization clients may additionally register to the back end to receive new data in real time via a WebSockets interface.

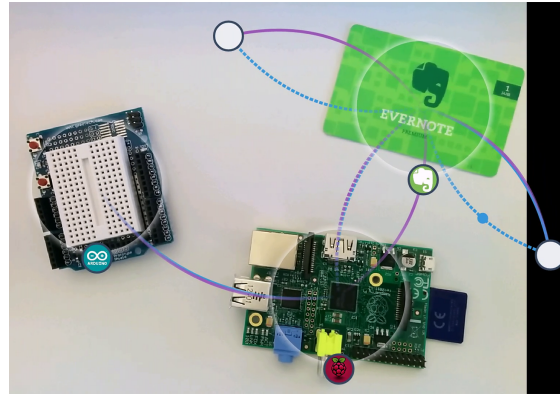
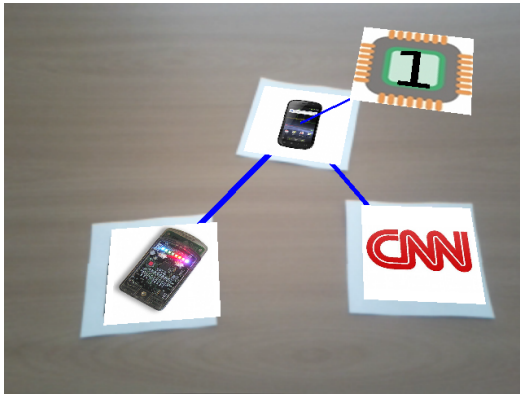
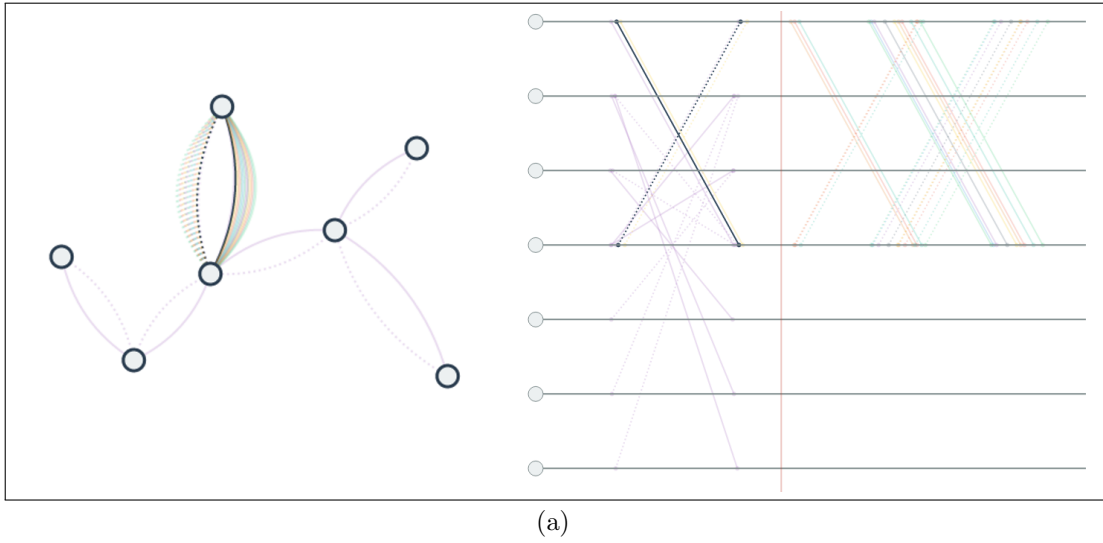


Figure 5.4: Different ways of visualizing the collected interaction information: Our Web application displays interactions between clients and smart devices in a force-directed graph and on a timeline (a). Our augmented reality interface overlays interaction information on the camera feed of a mobile device, either using fiducial markers to recognize objects (b) or using visual object recognition techniques (c).

We have implemented several different interfaces to display the collected data to end users who want to monitor device interactions: they can either use a Web application that visualizes the captured interactions as a force-directed graph and on a timeline (see Fig. 5.4(a)), or use an augmented reality (AR) application on a mobile device. The mobile application features two different modes of selecting devices whose interactions should be displayed – our initial prototype relies on fiducial markers that are attached to devices (see Fig. 5.4(b)) while another version makes use of the visual object recognition techniques presented in Chapter 4 to recognize smart devices in the field of view of the camera of the mobile device and then associates the recognized devices with nodes in the interaction graph and displays interactions between them as an overlay on the camera view (see Fig. 5.4(c)). In the following, we discuss all visualization methods, beginning with the Web application.

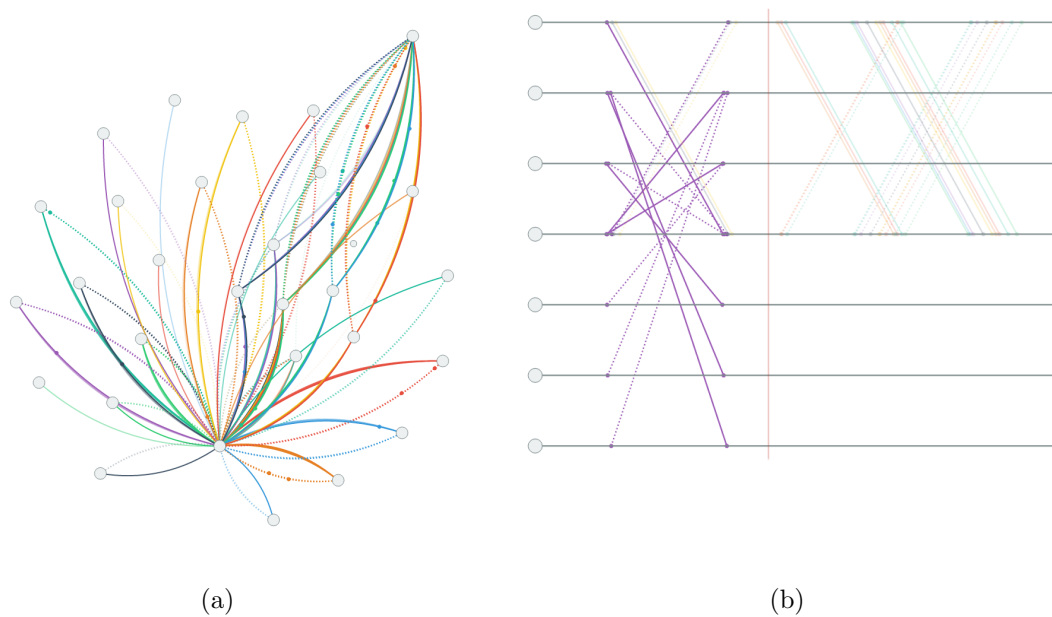


Figure 5.5: (a) Thanks to the underlying force-directed graph, the graph view of our Web application can visualize large numbers of interacting devices (image from [137]). (b) The timeline view of our Web application. The highlighted interactions (in blue) all form part of the same interaction chain.

5.3.1 Web Visualization Interface

To visualize logged interactions between devices, we implemented a HTML5 application that allows end users to inspect Web interactions between devices and remote services in real time and to replay recorded interactions. Our Web application provides two visualization modes – a graph view and a timeline view – as well as a split view that displays both modes next to each other. Because the travel time of a message in a local network is only a few milliseconds and users would thus not be able to trace interaction chains, our application virtually increases the latency of each interaction to one second for the purpose of the visualization. Consequently, using the vector clocks that give a consistent causal ordering of interactions between devices, our application delays the visualization of some messages – still, the start of each new interaction chain corresponds to the actual beginning of that chain, in real time.

Graph View The graph view of our Web application renders and animates a dynamic force-directed graph where devices or services are represented as nodes and interactions between them are visualized as dots that travel on the graph’s edges – the graph is aware of interaction chains, as illustrated in Fig. 5.5(a). Interacting nodes exert an attractive force on each other and unrelated nodes are repelled. The graph topology is updated on each frame with the most recent interactions – nodes that do not participate in any interactions gradually disappear from the graph.

Timeline View The timeline view visualizes Web interactions between devices in chronological order by connecting send and receive events on different devices that are arranged along the y-axis of the timeline (see Fig. 5.5(b)). This view can be used to inspect the exact order of the interactions. In contrast to the graph view, however, it does not provide any information about the topology of the communication.

Inspection and Filtering Apart from having the application display interactions between Web endpoints, users can inspect details of each interaction (HTTP headers, status codes, and message payload) by clicking the corresponding edge in the graph and timeline views. They can also navigate to related messages using previous/next buttons. Finally, the interface allows users to configure filters that change the visibility or color of interaction chains with specific properties. These properties can be set by the user via a simple domain-specific language. For instance, to filter all requests from the “TV” node to the `google.com` endpoint, the user would use this expression:

```
Request(from=TV)->Request(to="google.com")
```

5.3.2 Fiducial Markers-based Augmented Reality Interface

The described Web interface gives a convenient overview of interactions between devices, but fails to achieve our goal of presenting device interactions using a “magic lens” metaphor. As a first step to enable this, we created a mobile application that can augment the camera view of handheld devices with an overlay of live connections data. This application tracks networked devices in the live camera feed of the handheld device using fiducial markers and uses the sniffer application to obtain connection information about these devices. It then renders this information within the camera image as an augmented reality overlay: When a marker is recognized in the camera image, the picture of the associated device is superimposed over the marker. If markers of multiple networked nodes are recognized and our back end indicates that the respective nodes have been communicating with each other in the past, the devices’ pictures are connected by a line in the augmented camera image. If a node has connections to other nodes whose markers are not visible in the camera image, their pictures are displayed as hovering above the node to make the user aware of the connections. Multiple connected devices are arranged in a circle around the central node. If no pictures are available, the application uses a single placeholder image that represents multiple devices – in this way, we avoid overloading the viewport with too many images and rather emphasize those nodes that have been registered by the user.

Fig. 5.6 shows our demonstration setup that involved a Sun SPOT wireless sensor node with a Web interface and a mobile phone that both carry a fiducial marker. Our augmented reality application is deployed on a tablet and all devices are connected to an OpenWRT router with Internet access that is running our sniffer application. For Figs. 5.6(a) and 5.6(b), we deactivated all background update and synchronization services on the mobile phone and used it to access the Web interface of the wireless sensor node and to browse the `www.cnn.com` website (represented by the *CNN* logo). These connections

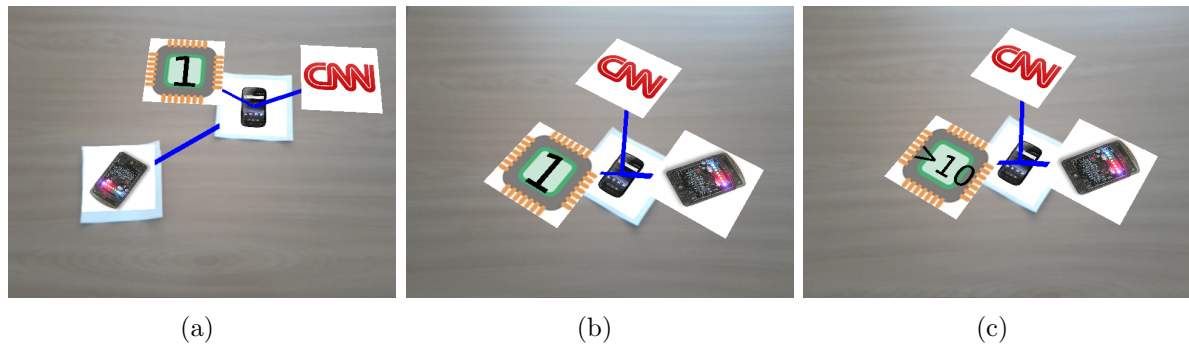


Figure 5.6: (a) Augmented Reality overlay of connections between the Sun SPOT, mobile phone, and CNN website. (b) The same situation without the Sun SPOT's marker being visible to the tablet's camera. A placeholder image shows that an additional connection exists, in this case to the nameserver used to resolve `cnn.com`. (c) The same situation with the mobile phone's background synchronization services enabled. Services without registered pictures are aggregated using a placeholder image.

are recorded by the sniffer application and relevant data about them is passed to the tablet device that identifies and tracks the markers and overlays the information about network connections on the camera image.

Fig. 5.6(a) shows a screenshot of the tablet application with all markers present while the Sun SPOT's marker was removed and thus no longer visible to the tablet's camera in Fig. 5.6(b). As a consequence of the previous interactions between the devices, the mobile phone is shown to have been connected to both the Sun SPOT and the `www.cnn.com` website. The line connecting the mobile phone and the Sun SPOT is thicker because of the stronger connection between these devices (i.e. because the Sun SPOT was accessed more often). Additionally, a placeholder image is shown as being connected to the mobile phone, which indicates that the phone had established a connection to one additional server which the application cannot provide a visual representation of. In this case, this unknown endpoint is the name server used to resolve the `www.cnn.com` domain. Enabling the mobile phone's background synchronization leads to many more such connections – to avoid overloading the viewport of the tablet device, multiple connections to unregistered endpoints are represented as a single placeholder icon (see Fig. 5.6(c)).

5.3.2.1 Augmented Reality Implementation

To recognize and track devices in the camera feed, we have experimented with two publicly available toolkits for augmented reality applications: Qualcomm Vuforia [320] and ARToolKit [249].

Vuforia is a free augmented reality software development kit from Qualcomm that is available for a variety of mobile operating systems including Android. Vuforia provides a comprehensive API and comes with a good documentation and a lot of example projects. Our tests have shown that the toolkit runs very stable also when tracking multiple markers. On the downside, its source code is not publicly available which hinders extensions and customization of the SDK. The greatest drawback of the Vuforia SDK, though, is that

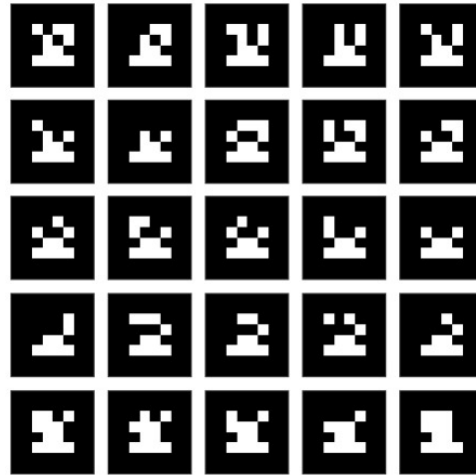


Figure 5.7: Examples of augmented reality markers that carry unique numerical IDs and can be used by ARToolKit (from [59]).

important parts of the image recognition are done on Qualcomm servers which makes any application using the toolkit highly dependent on that company. Finally, developers need to agree that usage statistics are sent to a remote server and the toolkit only supports a maximum of 512 markers.

ARToolKit for Mobile from ARToolWorks is the free mobile version of the open-source ARToolKit augmented reality software development kit. ARToolKit can track multiple markers and performs all image processing on the mobile device, which is the main reason for us to choose this platform for our application. ARToolKit can track 2D codes and general user-defined image markers. A strong black border is required around the markers to support the marker detection algorithm. The major disadvantage of allowing user-defined images as markers is that having a large number of such images drastically reduces the recognition performance as a detected marker must be compared against all registered images. For this reason, the toolkit also allows specialized *ID markers* that do not contain a user-defined image but rather a binary 2D array with a resolution of either 9, 16, 25, or 36 pixels (see Fig. 5.7). As the markers have to be rotation-invariant and thus at least three pixels need to be fixed to compute the marker's orientation, there is a maximum of 2^{n*n-3} unique IDs for a grid of size $n * n$ (i.e., 64 unique IDs for a 3x3 grid, 8192 for a 4x4 grid, etc.). Following the recommendation by the toolkit developers to use either 3x3 or 4x4 grid sizes, we have configured our application to use markers of size 3x3 which we generated using an online tool [322]. In our mobile application, such markers are associated to devices and used to identify and track these devices in the AR view. Using ARToolKit, we process camera images in five main steps:

1. Conversion of camera image to black and white to enable detection of marker borders and locate the marker
2. Calculation of marker position and orientation relative to the camera
3. Extraction of marker ID
4. Realignment of camera scene according to coordinate system of marker using the transformation matrix determined by step two
5. Execution of drawing commands in the scene

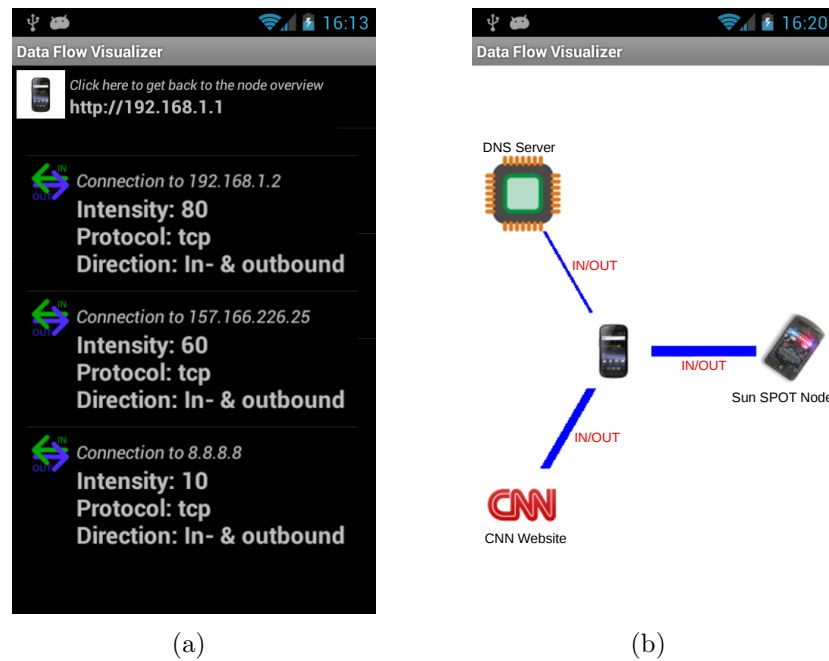


Figure 5.8: Alternative representations of the connections metadata: (a) The *text view* of the application shows information about the connections of a selected device in textual form. (b) The *graphical view* displays the connections using an abstract graph-like representation.

5.3.2.2 Obtaining Connection Data from the Passive Sniffing Back End

The augmented reality interface resolves AR markers of devices to their IP addresses (this information must be given by the user when scanning a new AR marker for the first time). These addresses are subsequently used to query the back end about further information on the identified device – the back end delivers connection information such as contacted IP addresses, protocols, connection strength, and traffic direction (inbound vs. outbound connections) as well as other data such as a node’s human-readable name, a description, and a picture. To be able to provide this information, the user has to manually register newly encountered tagged devices by scanning their AR-tag and providing their name, picture, etc. Whenever an AR marker is encountered for which a look-up at the back end yields no further identity information, a reverse DNS look-up is attempted to resolve the host name of the device, as this is assumed to be more meaningful for human users.

5.3.2.3 Discussion

Our evaluation within the demonstration setup showed that the proposed system works and that the mobile application and its augmented reality view is responsive enough to display connection information between devices and remote services in real time. Using ARToolKit to track the markers is stable, where we have tested the system with up to four simultaneously tracked tags without performance problems. When tracking more markers, we observed several crashes of ARToolKit that are most likely due to the toolkit running out of heap space. Furthermore, the camera of the handheld user interface device has to be brought rather close to the marker (about one meter) to identify it correctly,

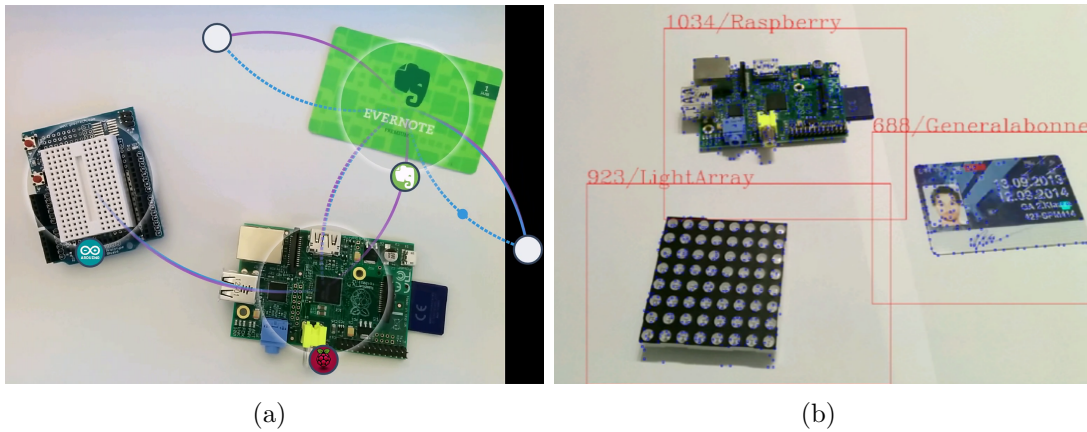


Figure 5.9: (a) Our application visualizes HTTP messages that are exchanged among devices as dots that travel along arcs between the objects. It uses visual object recognition techniques to identify devices in the camera view of the user interface. (b) The application can recognize multiple objects on a plain background using spatial clustering of FAST features.

which reduces the screen real estate available for displaying connection information and metadata. Finally, the battery lifetime of the mobile device running the visualization application is reduced considerably as AR operations are costly in terms of system resources.

Alternative Visualization Modes Apart from visualizing connections between end-points using the augmented reality overlay described above, our application prototype offers two other ways for users to access connections information: it can summarize all incoming and outgoing connections of a selected node textually (see Fig. 5.8(a)) and draw an abstract graphical representation of the connections (see Fig. 5.8(b)). To select a networked smart thing, the user can either scan its QR-code or point the camera of the handheld device at the AR marker of the smart thing, both of which yield the IP address of the device that can be used to query its connections metadata at the back end.

5.3.3 Visual Object Recognition-based Augmented Reality Interface

Recognizing devices based on fiducial markers has several disadvantages, most prominently the requirement to attach these rather conspicuous tags to smart things for enabling our system to track them and display their interactions. For a second prototype application that shares the goal of intuitively visualizing device interactions for human users, we therefore chose to use the visual object recognition techniques discussed in Chapter 4 as a means of identifying devices in the camera view of the user interface. This application is able to recognize up to four devices from a pre-trained set of about a dozen objects on each camera frame, associate the recognized devices to URLs and contact our logging back end to obtain interactions metadata – to obtain this information in real time, it registers with the back end via HTML5 WebSockets. The application then visualizes HTTP messages between devices as an augmented reality overlay (see Fig. 5.9(a)).

Multiple Object Recognition For the device recognition, our application proceeds similarly to the system described in Section 4.5.1: it first resizes camera frames to a resolution of 320x240 pixels and then applies an ORB feature descriptor to detected features. To separate objects, we spatially cluster keypoints using the DBSCAN [55] algorithm with a search radius of 30 pixels and extract ORB descriptors for each keypoint within a cluster's bounding box (see Fig. 5.9(b)).³ The object classes are described in the Bag of Words model and binary SVMs are trained for each class. At runtime, the mobile application applies brute force matching in the descriptor space using Hamming distances [71]. The SVMs determine the most probable object classes visible in the camera image. For the clustering of the multi-object recognition, it is necessary that the objects are sufficiently textured (i.e., contain at least 40 keypoints) and that there is little background clutter. If these conditions are met, our application achieves a frame rate of about 3 FPS when simultaneously recognizing six devices on a Nexus 5 smartphone, and up to 5.3 FPS when recognizing only a single device in the camera frame.

5.4 Applications

As already set forth in the motivation to this chapter, we envision the systems presented in this chapter to be applicable in a number of use cases that range from enabling inhabitants of smart homes to track what kind of information is leaving the domestic environment, and where control commands to their devices come from to visualizing interactions of different components of an industrial manufacturing line. Perhaps, the proposed system could even be helpful to educate students about distributed algorithms and for debugging distributed systems. In this section, we discuss several real-world use cases that illustrate the merit of our proposed approach to visualizing Web service interactions. We discuss another application of our system – monitoring interactions with Web-enabled automobiles – in the context of a case study about interacting with smart cars in Chapter 8.

Smart Homes Providing smart home inhabitants with a tool that enables them to stay aware of data that enters and leaves their home, as well as of interactions between smart devices within their domestic environment represents the original motivation for the work presented in this paper. Our system can clearly facilitate the monitoring of devices that handle privacy-sensitive data for end users, for instance with respect to smart electricity meters [188] – to find out which remote endpoints are accessing the domestic smart meter using our tool, users are merely required to point the camera of their handheld at the device and observe the overlaid interactions (see Fig. 5.10(a)). Using the Web interface, users can additionally examine individual messages and play

³Note that the FREAK (FAST) detector that was shown to yield comparable or even superior results to ORB in Section 4.4 is not well-suited for this application. This is due to FREAK depending on many more keypoints than ORB which makes it more sensitive to background clutter: because more features are extracted and described, DBSCAN detects more clusters, thus strongly decreasing the performance of the algorithm and yielding false positives. In our tests, we observed the algorithm extracting up to 15 clusters in situations where ORB only detects a single one, which strongly increased this method's processing time of the entire frame, in some cases to more than 1300 ms.

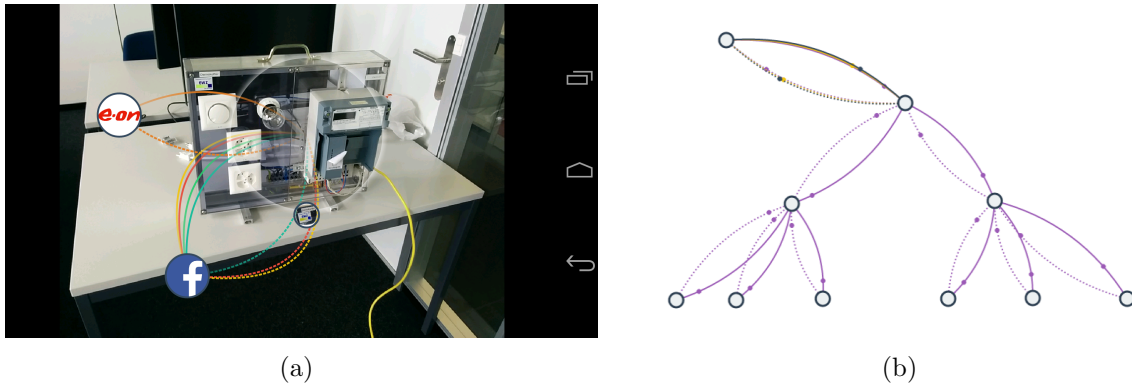


Figure 5.10: (a) Screenshot of the AR interface of our tool that visualizes interactions in smart homes in real time, for instance between a smart meter and utility companies or other parties that handle personal data. (b) Visualization of interactions of network endpoints that execute a distributed wave algorithm.

back earlier interactions between the domestic device and remote servers. Our system also enables users to monitor control commands that are sent to devices in their smart home, for instance to smart thermostats that are controlled by a remote service – this is especially relevant in scenarios where the control logic of appliances is provided by cloud services, as stipulated by the “Thin Server Architecture” paradigm [107]. Finally, our tool helps to monitor interactions *between* devices in smart environments – while this is not demanded by users at the moment, we believe that the accelerating deployment of home automation solutions will make it increasingly relevant to track such “behind-the-back” communication, especially when considering current research in the Web of Things domain that targets the ad-hoc creation of physical mashups based on user goals [139]. In such scenarios, users should be provided with a tool that helps them to monitor and manage interactions between devices, if only to give them a feeling of being in control of their smart home.

Smart Factories Production processes in factories increasingly involve dynamic interactions between individual manufacturing devices, to enable rapid reconfigurations which allow the process to evolve and adapt to the mass customization of products [111]. The technologies proposed in this chapter could support operators within such environments to rapidly determine which devices talk to each other and what data is transmitted between them at any given moment. This is especially helpful when devices that are involved in a process start to be aware of their functionality, thus enabling the dynamic reconfiguration of the system at run time (see Chapter 7).

Smart Firewalls With respect to both, smart homes and smart factories, we believe that it could be beneficial to combine the systems presented in this chapter with network infrastructure that can be remotely configured by users. In this case, if users discover an interaction between devices or with a remote service that they feel uncomfortable about, this connection could be instantly terminated and prevented in the future literally at the fling of a finger by configuring a firewall rule at a network router. We believe that this

combination of software-defined networking with our augmented reality interfaces that display interactions holds great potential to facilitate not only network monitoring but also the *management* of networks at home and the workplace. As a proof of concept, we implemented this functionality in a private home network – however, because we were unable to configure the firewall rule directly on the Zyxel NBG5715 router present in that network, we created a simple wrapper for the `iptables` program which allows remote users to set local firewall rules by specifying the IP address of the device to be denied access to a smart thing. With the help of this program, users of our visualization application are now able to define basic policies that govern which devices are allowed to communicate with which endpoints within a home network.

Smart Debugging and Education We believe that our system could be applied in educational environments, similar to visualization tools for algorithms and data structures. Several studies in the domain of algorithm visualization have shown that students believe such tools support them in learning rather abstract programming methods such as sorting or line sweep algorithms, and examination results also reflect these benefits [98]. By visualizing interactions between distributed agents, our tool extends the scope of the broad domain of software visualization [183] beyond individual programs. This is, for instance, relevant with respect to advanced topics in the domain of distributed algorithms (e.g., regarding leader election, distributed termination detection, and with respect to general wave algorithms, see Fig. 5.10(b)). Our system can be used to illustrate these because it also takes care of preserving the causal relationships within interactions that stretch over multiple exchanged messages – at the same time, the tool itself represents a very convincing example to demonstrate the benefits of vector clocks in classrooms. We recommend using the Web visualization interface for these use cases – potentially, however, the magic lens interface of our visualization tool can be used in educational settings as well, for instance in robotics courses.

5.5 Summary

In this chapter, we presented a system that allows to visualize interactions between networked smart things using different techniques. Among these is an application that visually recognizes devices in the field of view of the camera of handheld or wearable devices and renders an augmented reality overlay to visualize interactions between them and with remote services. To record the necessary communication metadata, we proposed two different systems, the first of which has the advantage of not requiring any specialized network infrastructure and, indeed, no changes to the networked devices themselves. This passive sniffing system was successfully deployed on a commodity router platform that is present in many private homes, but fails to deliver the in-depth information about network interactions that we seek to display to users. Therefore, we also proposed an active logging system where logging clients are deployed on the communicating smart things and services themselves and report their interactions to a central back end. Although this approach requires system components to be modified, we succeeded to keep these

modifications to a minimum, by using Java agents to rewrite bytecode just before its execution.

We envision such a system to be deployed in smart homes, where it would enable end users to monitor communication patterns of networked devices inside their home and with services in the cloud. By using the application, inhabitants could keep track of which of their smart devices interact with each other and whether a device inside their home uploads possibly privacy-sensitive data to remote services or receives control commands from the outside. Other application areas that we believe such a system to be useful for encompass the monitoring of interactions in smart manufacturing scenarios and training and debugging of distributed systems and algorithms.

CHAPTER 6

A Web-based Infrastructure for the Internet of Things *

In the previous chapter, we discussed how Web technologies and visual object recognition methods could be combined to make humans aware of device interactions in a smart environment. This is important especially in the context of smart homes because it can help inhabitants trust their smart devices, ultimately putting them in control of what information their smart things transmit to the outside world, and what commands they receive. In this chapter, we discuss another fundamental issue that must be dealt with to make the Web of Things widely usable: making it possible for computers and human users to *find* services that are provided by smart things.

Up to this point, we implicitly assumed that each service in a smart environment is embodied in a specific object that can be selected using technologies such as NFC, by using fiducial markers, or via the visual object recognition methods that we detailed in Chapter 4. We now drop this assumption: a client that wishes to use a service provided by a smart device should no longer be bound to having direct physical access to this device for selecting it, but should rather be enabled to find and use the service from anywhere, at any time. By the same token, our discussion now also encompasses services that are not embodied in physical objects and therefore cannot be selected by pointing, scanning, or touching in any case. These services are “virtual” by their very nature and sometimes do not even directly influence our physical environment, for instance in the case of translation services.¹

In this chapter, we present a mechanism that makes services in smart environments searchable in a reliable, fast, and user-friendly way. In the “traditional” Web that is composed of static documents, and the “Web 2.0” that focuses on the importance of user-generated content, the task of making billions of documents findable was handled by Web search engines such as Yahoo! or Google. However, making smart things and their services findable is significantly more complicated than enabling the searching for

*This chapter is based on the following published articles: [129, 134, 136]

¹If a user wishes to, such virtual services can of course also be assigned to physical objects, thereby enabling the techniques for device selection that we discuss in Chapter 4 for them as well: for instance, a user could choose to associate a language translation service with a physical dictionary and access the service via that object.

documents for a number of reasons [197, 220, 136]: First, services provided by smart devices often should be identified based on dynamic, contextual information such as their location or their current state. Second, there is no uniform and widely accepted way of describing smart things and their services. Third, clients of services that are provided by smart devices in many cases are computer programs themselves and therefore require descriptions that are provided in a format that is easy to parse and interpret for machines. Furthermore, a machine-readable description format for smart devices must also include a mechanism that allows machines to find out how the service can be used (i.e., information about its machine API) – information provided for devices is thus different in kind from information that is targeted at human users, and is not necessarily expressed in a way that is easy to index for traditional Web search engines that are geared toward finding textual documents which were created for humans.

For these reasons, before smart things and the services they provide can be made findable, a method is required that allows search engines to obtain the above-mentioned information about their context, dynamic properties, and service interface. This process, that we refer to as “resource discovery” [226], takes place after the network presence of a smart device has been established and an entry point – such as an authoritative URL or hostname – to its services has been found. A search engine that can discover and index services in this way would not only be able to allow humans to interact with smart things by enabling them to find and use the data and functionality they provide, but would do the same for machine clients, thereby allowing them to support users in finding services and even to use these services themselves, on behalf of users. From the user perspective, the main benefit of discovery and search mechanisms for the WoT is a major simplification of human-machine and machine-machine interaction in smart environments that gives humans greater power to configure and control ubiquitous smart devices.

Apart from facilitating the interaction with the WoT for humans and machines, a search engine for smart things and their services must overcome another major challenge that is associated with the expected very large number of networked devices in future smart environments: according to many, the Internet of Things is expected to have a much larger overall scope than the Internet of computers [127]. This renders a centralized solution that enables the discovery and searching for smart things undesirable, if not entirely impossible, and calls for designing a WoT search engine in a way that can scale to billions of connected devices while enabling smart things to cooperate on a global scale.

In the following, we first discuss several approaches to the description of smart devices as well as infrastructures that aim at administering large numbers of smart things in IoT scenarios. Next, we describe our own proposal for a Web-based management infrastructure that makes smart devices and their services searchable and, in particular, discuss its hierarchical structuring and its resource discovery and look-up components. This infrastructure has been used in multiple projects to facilitate the usage of services in smart environments for humans and machine clients. It also represents a cornerstone of a method that we propose to enable smart devices to cooperate automatically, by sharing functional semantic metadata about their capabilities – this system, and the role of our management infrastructure in its context, is discussed in detail in Chapter 7.

6.1 Middlewares for the Internet of Things

The challenge of making services in an ecosystem of billions of devices searchable and findable is part of what has been termed the “Findability Layer” of the WoT [73]. Overcoming this challenge is important in the IoT domain, most prominently because we expect that a reliable and usable search engine for smart things will allow us to realize the dream of being able to “query the real world” – and thus enable us to look for smart devices and their services in a way similar to how we use traditional search engines to locate information on the Web. Furthermore, such a service is necessary to enable the composition of services in smart environments within physical mashups that provide higher-level functionality than their individual constituents [6, 139].

Consequently, to make smart things findable and easily usable for clients, many infrastructures and middlewares that manage devices in smart environments have been proposed. Examples of very early, groundbreaking projects are Hewlett-Packard’s CoolTown [45, 101] and the E-speak project [61]. More recent research initiatives range from projects by individual research groups (e.g., [6, 36, 217] as well as our own projects [136, 225, 226]) to massive international endeavors, for instance in the context of the Seventh Framework Programme of the European Commission (e.g., the SMART [265] and IoT-Architecture [268] projects). Often, however, management infrastructures that provide a search mechanism for smart devices were created for specific use cases, most prominently approaches that facilitate the publishing and consumption of data in sensor networks [96, 178], or have been tailored to the specific mode of interactions between devices, for instance emphasizing device mobility in dynamic environments [221].

A survey of many of the most prominent IoT middlewares is presented in [9] – the authors classify the diverse systems according to the functionality they provide: each of the considered infrastructures provides a way of managing devices internally (including device discovery), but only a few of them are compatible regarding their programming platform. For instance, HYDRA [110], and ASPIRE [264] follow the OSGi platform model while the SIRENA [22] and SOCRADES/SOA4D [267] middlewares use a DPWS-based service integration system. This formation of islands on the level of IoT middlewares does, in our opinion, hinder the proliferation of the IoT as a whole.² Some even argue that, since none of these infrastructures is without its shortcomings and thus none will perhaps prevail over the long term, we will witness the creation of ever more (incompatible) platforms in the future [169]. The situation is similar in industry, where standards have been proposed that target specific use cases – an example for this is the system created by the Digital Living Network Alliance (DLNA) [260] that aims at establishing interoperability between multimedia devices.

In the design of our own management infrastructure that we propose in this chapter, we fully embrace the application-level convergence and interoperability put forward by the Web of Things vision: our main goal is to make full use of Web standards by providing uniform interfaces that are easy to understand for clients both when searching for smart things and when registering their own devices and services, and that additionally facilitate

²This is also one of the main arguments brought forward by the FP7 IoT-A project [268].

the application-level integration with other middlewares. We have shown that, due to the adoption of Web standards, our infrastructure is compatible with projects by other research groups, for instance from the body area networks domain [205], and with other middlewares that target event-based interactions between smart devices [40], or aim at semantically enriching smart things [68].

6.2 Finding and Describing Smart Devices

Before smart things can be indexed and made findable using a look-up service, there must be a way of how this service can obtain information about devices and their capabilities. Indeed, the description and discovery of services have been identified by some as two of the four main research challenges in the “Future Internet” [91] which, in this context, refers to the Internet of devices and services (the other two challenges relate to service access and service composition). In particular, the authors of [91] highlight the importance of future IoT systems being able to handle heterogeneous service descriptions.

Describing services that are provided by smart things within the WoT is very similar to the more general challenge of providing machine-understandable specifications for Web services. To describe these, many languages and frameworks have been proposed in the past – in the context of the WoT, the most prominent include Microformats [304] and Microdata [331], both of which represent ways of enriching HTML documents with machine-readable metadata to make the services that are represented by these documents discoverable for humans and machine clients [129]. We believe, however, that discovery services should not be constrained to only considering specific ways of embedding service information – rather, we propose a discovery service that remains extensible regarding future description formats (see Section 6.3.2).

Apart from the representation format of service descriptions, it is also important to determine what kind of information should be part of a service description. For the WoT domain, the *Smart Things Metadata (STM)* model has been proposed [73, 77] which specifies the contextual information and metadata that any infrastructure should be able to obtain from a smart thing to make it searchable. This model has been created while considering other approaches to describing smart things [2, 23, 48, 92] as well as our own experiences when designing earlier versions of our management infrastructure. It incorporates two types of information about a smart device: *static properties* such as a description of a device and its services, and *dynamic characteristics* that are related to the object’s context and to quality of service parameters.³ This model represents a recommendation of what kind of information should be provided by smart things (and, thus, should be considered when creating description frameworks) but is not exhaustive – nor is it mandatory that a device provides every bit of information that is specified in the format. Rather, we envision search infrastructures for smart things to mimic traditional Web search engines in that they should aim at interpreting all available information about a Web resource to be indexed [73, 134].

³We do not reproduce a diagram of the STM model in this thesis – the complete model can be found in [73].

Given past experience, we are skeptical that the proposed STM model – or indeed any currently proposed guideline of what information should be contained in Web service descriptions – will become the predominant method of describing smart devices and their capabilities. For this reason, focusing on the *interoperability* of discovery services has been defined as an important area for future IoT research in [91] and frameworks that can handle heterogeneous service descriptions have been proposed also in the context of other middlewares, for instance in [155] and [186]. Likewise, we also propose that our management and search infrastructure for smart things should not be tied to a single description format but rather make use of an extensible discovery mechanism that incorporates many of the above-mentioned strategies for discovering services. Similar to other recent projects in the domain of Web service discovery, our discovery service has been created with broad interoperability in mind, and is thus not tied to our management infrastructure. Rather, it provides “Discovery as a Service” [54] for any client that requires this functionality via open Web APIs. In Section 6.3.2, we introduce this service, and in particular discuss its ability to be extended at runtime with new resource discovery strategies. Each of these strategies handles one of the description frameworks that have been proposed for smart things (e.g., Microformats or Microdata) by mapping it to a common internal description format. Our system can also publish its internal descriptions in many formats, thus making it a universal translation framework for smart things metadata.

Already at this point, we mention that the challenge of identifying and matching Web services has also attracted much interest from the Semantic Web community – a central argument here is that, by basing descriptions of Web services (and, by extension, of services that are provided by devices in the WoT) on semantic technologies would not only facilitate the discovery of these services but also their automatic behavioral control, and enable the automatic composition of device functionality to yield higher-level services [99]. Consequently, the Resource Description Framework (RDF) [334] has been proposed as a general approach to describing Web service capabilities, most prominently in the context of describing sensors and sensor systems using SensorML [23]. We discuss the usage of semantic technologies for describing the characteristics and capabilities of Web services, as well as of approaches from the domain of service-oriented architectures (e.g., the *Web Services Description Language*, *WSDL*) in greater detail in Chapter 7, where we focus on the semantic interoperability of Web services. For now, we focus on best-effort discoverability and searchability of WoT devices and their services via the more conservative method of embedding structured metadata.

6.3 A Web-based Infrastructure for Smart Things

In this section, we present an implementation of a distributed hierarchical Web-based management infrastructure that can discover smart devices, interpret service descriptions that they provide, and make the services searchable for clients. Similar to other infrastructures that enable the searching for services provided by smart devices (e.g., [77, 193]), our infrastructure includes three core components: a way of discovering and interpreting resource descriptions, a resource repository that stores discovered descriptions, and

a search component that is responsible for the service look-up itself. After introducing the major requirements that our management infrastructure shall satisfy and discussing several design decisions that we made in response to these, we describe the discovery and look-up components of the system in greater detail.⁴ Regarding the discovery of smart devices, we in particular discuss the run-time extensibility of our universal discovery service, while our presentation of the look-up service will focus on its treatment of *queries as REST resources*, which enables automatic load-balancing across the infrastructure, and allows for an advanced query caching mechanism that reduces the time and number of messages required for service look-up.

6.3.1 Requirements and General Design Principles

As already mentioned, we want our proposed management infrastructure for smart devices to be *interoperable* with other middleware solutions to avoid creating an island solution. We aim to achieve this goal, as well as a high degree of *user-friendliness*, by having all client-facing APIs of the infrastructure adhere to widely accepted Web principles such as uniform interfaces, thereby making them simple to use. The infrastructure should also be able to administer the expected high number of digitally augmented devices in future smart environments: this *scalability* shall be achieved by a distributed, hierarchical structuring of its individual nodes. Finally, because the adoption of a decentralized structure impedes the simple management of our infrastructure, its nodes should exhibit a certain degree of *self-management*, which includes the ability to recover from individual node failures. We discuss each of these goals in more detail in the following.

6.3.1.1 Interoperability and User-friendliness

As indicated in the previous section, we aim to implement the discovery and look-up components of our management infrastructure in a way that is “future-proof” regarding prospective developments in the IoT domain and, more broadly, with respect to future methods of describing Web services. Therefore, we have adopted a requirement for interoperability as the most important principle in the design of this system: the management infrastructure should be interoperable with other search engines and description formats. To satisfy this requirement, our system makes use of an extensible universal discovery mechanism (see Section 6.3.2) and provides a uniform REST look-up interface for clients that search for smart devices and their services (see Section 6.3.3). Responses to queries to that interface are given in a format that can again be translated into many broadly used device description formats using our discovery service. Thus, the discovery component is central to the design of our management infrastructure – still, this service can also be used by clients directly, as it is an external, standalone, component of the proposed ecosystem. Closely connected with the requirement for interoperability but shifting the emphasis to human users of our infrastructure, we demand that its interface be easily understandable and usable not only for machine clients but for people as well. This requirement will be

⁴The implementation of its resource repository and an earlier version of its discovery service is based on work that was carried out prior to this thesis [128, 226].

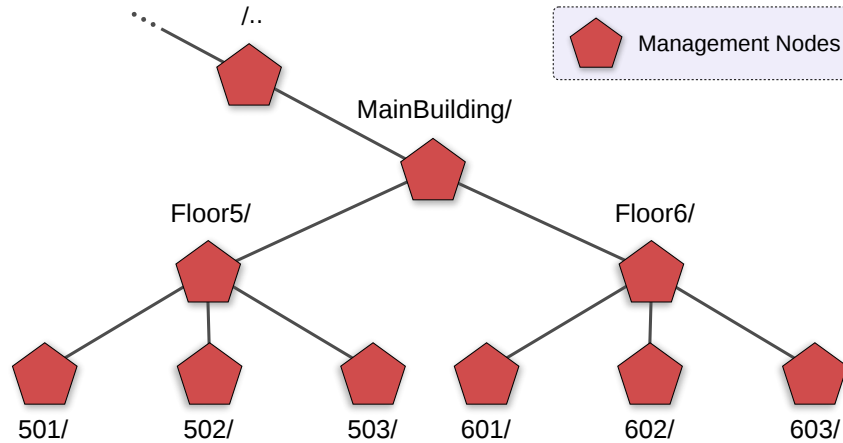


Figure 6.1: The nodes of our proposed management infrastructure are structured hierarchically according to logical place identifiers. In this example, nine nodes are spread across two floors with three offices each in the building *MainBuilding*.

reflected in the design of its look-up service that can not only handle structured queries from machines, but also unstructured, keyword-based, requests.

6.3.1.2 Scalability

Another major requirement on our management infrastructure is that we require it to specifically address the challenge of scaling to huge numbers of connected interacting smart devices that produce large amounts of traffic in IoT environments. It will most probably not be possible to utilize traditional centralized approaches to service look-up in smart environments that are populated by billions of smart devices [270] – rather, we propose to sub-divide our search space hierarchically, where we use the *location of a smart device* as its key dynamic feature for segmentation [226]. This is beneficial because smart things interact much more frequently with other devices in their immediate environment than with objects that are situated at an entirely different location [238]. We can exploit this locality of interactions within rooms, floors, and buildings to limit the messaging overhead when administering devices at different locations and propose to arrange the individual nodes of our infrastructure hierarchically, where each node is responsible for a specific spatial domain. This approach allows to contain look-up operations to their relevant subtree and avoids the global routing of search queries within our infrastructure. Still, our system retains the ability for clients to search globally if they wish to do so.

Fig. 6.1 shows an example for the hierarchical structuring of our management infrastructure that we will refer to at multiple times throughout this chapter to explain how the discovery, and look-up of smart devices and their services works in our case: in the example, the infrastructure sub-divides a building into multiple hierarchically structured administrative domains, each of which is managed by an individual infrastructure node. For instance, the node that is responsible for an office has, as transitive parents, the node that administers the floor this room is situated on and the management node responsible for the entire building. “Management,” in this case, refers to the respective node storing information about services that are provided by smart things in its domain and taking

care of correctly routing received search queries (we explain how the routing works in more detail in Section 6.3.3). Purely virtual services that are not embodied by any device can in principle be registered to any node – still, it might be beneficial to use the look-up history for such services to determine the infrastructure node that is closest to their most frequent clients and moving information about the service to that location (or replicating/caching it there).

Note that, in our system, we restrict the direct communication between management nodes to direct interaction between neighboring nodes in the hierarchical structure (i.e., a node can only communicate directly with its parents and children). This means that individual nodes can remain ignorant about most others in the system: they need only know their direct neighbors for the infrastructure to perform its functions. This guarantees that the system remains scalable in terms of extending the hierarchy by adding more administrative domains. Furthermore, the interaction between two devices that are administered by the same management node only concerns that very node, and such devices can find each other without triggering a look-up that concerns any additional management nodes. Likewise, two devices in the same building can interact without affecting any management nodes that are responsible for places in the outside world.

Regarding the individual nodes, we and others have proposed that only management nodes that administer more fine-grained domains (e.g., rooms or floors) be “embodied,” for instance as wireless routers or network-attached storage devices [226, 227]. Such embodied nodes are, in addition to their tasks that relate to the administration of smart things in their domain, responsible for linking devices that do not directly provide Web connectivity to the rest of the system, i.e., act as *smart gateways* [226].

6.3.1.3 Self-management

The adoption of a distributed hierarchical structure as one of the main architectural principles of our management infrastructure brings with it a requirement to keep manual administration of the individual nodes to a minimum, for the system to remain manageable even in extensive settings. For this reason, we have embedded the ability for structural self-configuration in our system, meaning that only the assignment of nodes to their locations must be done by hand – given this indication of their administrative domain and the URL of a single common “top-level” node, all other infrastructure nodes are able to take their place within the hierarchy automatically. The same mechanism makes the infrastructure resilient with respect to individual nodes becoming unreachable: in this case, the system rearranges its nodes such that global connectivity is, again, guaranteed. When nodes recover from failure, they are again integrated into the system and the original configuration can be restored.

The proposed self-management of nodes is possible because the entire topology of the system is induced by the names of its administrative domains – only if all of them are assigned according to uniform naming conventions, we can guarantee the functioning of the routing algorithms: for instance, the same location identifier must not be used more than once within the infrastructure, and the names must be built in a way that reflects their hierarchic ordering (e.g., “Building/Floor/Room,” for instance in “Main-

Building/Floor6/602”). This is not straightforward, as users who set up management nodes in their private homes are not automatically prevented from calling the administrative domain of their node “ETH,” or even “Switzerland.”

Location identifiers that we use in our deployments are derived from a strictly hierarchical symbolic location model, meaning that the infrastructure nodes remain ignorant about whether these identifiers are associated to any absolute geometric location such as GPS coordinates. We decided to base our naming system on a centrally controlled vocabulary, since, although many location models have been proposed (e.g., [11, 16, 238]), there is, at the moment, no widespread standard of how locations – in particular, indoor locations – are modeled [225]. In principle, however, our system is capable of handling any location model that allows for location hierarchies – such models are the subject of current research and standardization efforts, most prominently by the Open Geospatial Consortium whose Geosemantics Domain Working Group [314] is working on creating a semantic framework for representing geospatial knowledge, specifically with a focus on the Internet of Things and the Web of Things [315].⁵

6.3.2 Smart Things Discovery

One of the two main modules of our management infrastructure is its discovery component that, given the URL of a smart thing, is responsible for gathering information about this device and the services it provides. The device discovery system of our infrastructure is, in fact, a standalone discovery service and exists detached from the individual management nodes. It can therefore also be used by other clients that wish to obtain information about a Web resource of interest, and as an on-the-fly translator for smart things metadata.

One of the biggest weaknesses that haunt software used to parse non-standard formats is that they become outdated as soon as amendments to current description schemes are made, or when these schemes are replaced by newer formats. For this reason, we created a service that can be extended at runtime with new strategies of parsing service descriptions and thus is future-proof with respect to new description formats. Our system allows developers to inject new methods of parsing service descriptions on demand, whenever they discover a resource that cannot be handled by any of the currently implemented strategies. In this way, this resource, as well as all resources that are described in a similar way, immediately become usable for all clients of the service.

To enable this behavior, the service analyzes the reference to a resource (e.g., the resource URL, an already downloaded resource representation, etc.) that a client sends as part of its query as well as information about the resource that can be obtained from the client’s query (e.g., by downloading representations of the resource in multiple different multimedia types) using all its currently registered discovery strategies (see Fig. 6.2). The information that it obtains from the individual strategies is then merged and consolidated, and used to create an internal representation of the resource. This information constitutes the answer of the service to the client request and can be transmitted in one of multiple formats (this is under the control of the client, using HTTP content negotiation).

⁵See [82] for an overview of current approaches to semantically modeling locations and a concrete example of such a model.

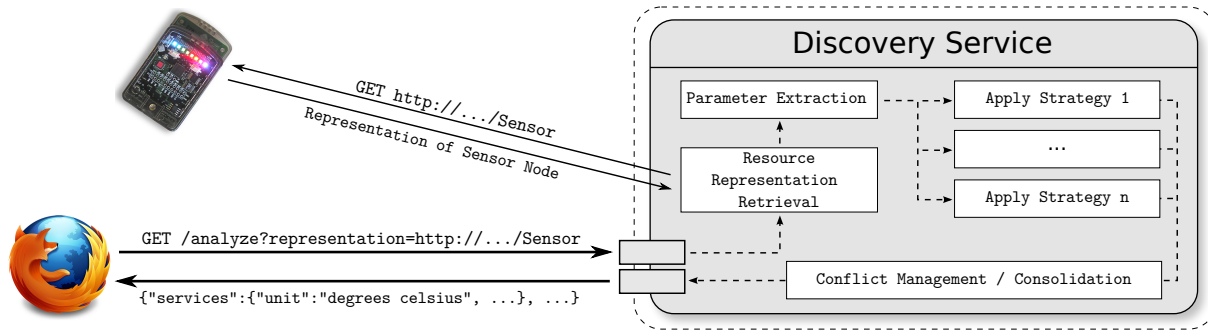


Figure 6.2: Interaction of a client with our discovery service: The client sends an HTTP GET request to trigger the semantic discovery of a resource. The discovery service uses information that is contained in the query to obtain representations of the resource in several different formats. Multiple discovery strategies are applied to these representations, their outputs are consolidated, and the resulting resource description is returned to the client.

The Web interface of our discovery service itself is based on REST – the main HTTP verbs map nicely onto its functions and provide an intuitive and user-friendly interface: GET is used to invoke the main function of the service, i.e., to obtain resource descriptions and to get information about registered strategies, each of which is represented as a Web resource. POST, PUT, and DELETE are used to create, update, and delete discovery strategies. The discovery service furthermore provides information about its own capabilities using Microformats markup and an OpenSearch description [247].

6.3.2.1 Internal Representation of Web Resources

As already mentioned in the introduction to this chapter, multiple models exist that attempt to capture the most important metadata about Web services. Considering a broad range of these models is crucial for our discovery service, since all information gathered about a service is merged into an internal service representation. However, the internal service representation constitutes a potential bottleneck concerning the extensibility of our discovery service: even if the service was able to parse every single way of how properties of Web services can be described, all of this information would still be consolidated into a common internal representation before sending it to the client. Therefore, if a description format incorporates information about a characteristic of a service that cannot be accommodated in the internal representation, this information is potentially lost during the consolidation phase.

We address this challenge in two ways: First, our internal description format is based on the STM model [73, 77] that we briefly described in Section 6.2 and that considers several different approaches to describing smart devices and their services [23, 92] as well as our own experience [2, 48]. Second, to guarantee the compatibility of our service with completely new device properties, our system allows not only the dynamic creation of new discovery strategies but furthermore is based on a dynamic internal resource description format, meaning that it is automatically extended using reflection when a new discovery strategy considers device properties that are not part of the STM model.

```
1 {
2   "name": "Temperature Sensor",
3   "provides": {
4     "result": "current temperature",
5     "unit": "degrees celsius"
6   },
7   "rating": 3.4,
8   "xyz": "Example"
9 }
```

Listing 6.1: JSON representation of a Web-enabled temperature sensor.

```
1 [
2   {"name": "Name"},
3   {"provides.result": "Services.Output"},
4   {"provides.unit": "Services.Unit"},
5   {"rating": "Review.Rating"},
6   {"xyz": "A.New.Identifier"}
7 ]
```

Listing 6.2: JSON document that a client sends to the JSON strategy stub of our discovery service for adding a new mapping that corresponds to the temperature sensor description in Listing 6.1.

6.3.2.2 Extension of Strategy Stubs

Our proposed discovery service enables clients not only to discover smart things metadata, but also to extend the service at runtime by injecting new ways of resolving representations of Web resources: clients can create new discovery strategies and also extend existing ones using the REST interface of the service. We distinguish between two approaches of extending our discovery service: clients may *create* completely new discovery strategies, and may *extend* already existing strategy stubs. The service already features such stubs for formats that are widely used to describe Web services, such as Microformats or Microdata, and also for descriptions that are based on the JSON and XML languages: whenever a client desires to add discovery capabilities that make use of one of these formats, it is sufficient to submit a new *mapping* between the service descriptions to be parsed and the representation that our service uses internally (this is explained in more detail below). As an example for our discussion of how clients can extend the discovery capabilities of our service, we use the simple JSON representation of a Web-enabled temperature sensor shown in Listing 6.1.

As briefly hinted at above, we consider a single discovery strategy to be a composite of a strategy stub and a corresponding strategy mapping. To make a Web service that is described in the same form as the temperature sensor in Listing 6.1 discoverable by our service, a client would extend its JSON strategy stub by submitting a corresponding mapping of the sensor's metadata format to the internal representation of the service. This is accomplished by sending a `POST` request to the Web resource which represents the JSON strategy stub with the content shown in Listing 6.2.

From this information, our discovery service creates an internal mapping from resource

```
1 {
2   "description": "Temperature Sensor JSON Schema",
3   "type": "object",
4   "name": {
5     "type": "string"
6   },
7   "provides": {
8     "type": "object",
9     "properties": { ... }
10  },
11  "rating": {
12    "type": "number",
13    "pattern": "[1-4]{1}[\.]?[0-9]*"
14  },
15  "xyz": {
16    "type": "string"
17  }
18 }
```

Listing 6.3: JSON Schema document that a client sends to the JSON strategy stub of the discovery service to add a mapping that corresponds to the temperature sensor description in Listing 6.1 (the service properties have been omitted for conciseness).

descriptions to its internal format where, for instance, a JSON `result` element that is nested in a `provides` element would be mapped to the `Output` identifier within the `Services` structure of our internal description format. The identifiers that can be used for mapping to the correct internal structure are exposed via the REST interface of our service which also gives provides description of the type of information that is contained in each of the specified identifiers. The identifiers correspond to the STM model that was described above and that we expect them to cover much of the metadata that Web service providers could wish to expose about their services. If, however, our discovery service encounters a new mapping that contains an unknown identifier as a target location, such as in the example given above (the `A.New.Identifier`-field), it uses reflection to add this field to its internal representation of resources.⁶ To summarize, by injecting a new mapping into an existing strategy stub, a client creates a new, purely syntactic, mapping of data fields in external descriptions to the internal description format.

While it is convenient for clients to inject new mappings in the way described above, this method is not suitable for every type of resource representation. For this reason, our service also comprises another mechanism for the extension of strategy stubs that uses data schemas (e.g., JSON Schema or XML Schema information) to create new mappings. In this case, the client would proceed in a two-step-process, first submitting a new schema that describes the service metadata, and then submitting the corresponding mapping to insert the correct syntactical mapping. To give an example, the JSON Schema document for the above temperature sensor description is given in Listing 6.3. The integration of a new mapping in this “schema-driven” way requires more client interaction with the

⁶We propose that newly created fields should be monitored by a privileged user and that they be pruned from the structure if deemed too application-specific in nature.

service. However, it enables clients to specify the type of the information in service descriptions as well as concrete permissible patterns for the contained data, as can be seen in Listing 6.3 for instance for the `rating` field of the service. This means that the discovery service is able to automatically discard values that do not correspond to the pattern or data type specified in the schema. This is not possible in the case of *schema-less* injection of mappings – in this case, the discovery service proceeds in a best-effort manner and guess the data type of newly created fields. We recommend that mappings which include submitting a new schema should be adopted where required, but that the simpler approach of directly injecting mappings should be preferred for formats that do not specify explicitly typed data (e.g., Microdata).

6.3.2.3 Creation of new Discovery Strategies

Our discovery service supports the creation of entirely new ways of parsing resource descriptions, by allowing clients to inject new discovery strategy stubs (in the form of Java classes) at runtime. To mitigate the major security risks that are inherent to this approach and to block malicious code from being inserted into the service, we suggest a semi-automatic injection mechanism, meaning that novel strategies are first reviewed by a privileged user.

6.3.2.4 Data Integration and Strategies Conflict Handling

Because our discovery service applies all registered strategies to device representations that are submitted by clients, conflicts between the strategies may arise: for instance, different parts of a single service description may correspond to different strategy matcher patterns, or the service might derive differing information from different representations of the same Web resource (e.g., if the resource provides an XML representation as well as a representation in the JSON format). After applying all strategies to a submitted description, our service attempts to merge all returned descriptions into a single instance of the internal resource representation. This merging is not straightforward when treating conflicting data for the same field: in this case, we use a confidence score that depends on the ratio of correctly matched fields in the resource representation (i.e., a measure of how well the individual strategies fits the resource representation) to break ties. To avoid conflicts altogether, our discovery service additionally provides clients direct access to individual strategies – in this way, the client can control which strategy is applied when analyzing a submitted resource representation.

6.3.2.5 Implemented Discovery Strategies

Our discovery service already contains strategy stubs that consider several description formats for Web services (a more detailed account of these is given in [129]). In particular, it can parse multiple lightweight markup languages that are used for describing Web services, such as Microdata and Microformats – the main properties of these formats are discussed below. The service also supports resource descriptions that are based on JSON, XML, and RDFa. The JSON format is particularly interesting as its importance in the

```
1 <div class="service">
2   <p class="label">ACME Hotels</p>
3   <div class="operation">
4     The operation is invoked using a
5     <span class="method">GET</span> on
6     <code class="address">http://example.com/h/{id}</code>, with
7     <span class="input">the hotel ID replacing <code>id</code>.</span>
8   </div>
9 </div>
```

Listing 6.4: hRESTS service description markup, adapted from [105].

Web domain has grown steadily over the last few years, starting in the year 2010 [261]. It is also easier to parse and typically has a smaller footprint compared to XML. We believe that the JSON format would be suitable as an interchange format to transmit information about Web services – our discovery service exposes its internal resource representation as JSON as well meaning that, because it contains a strategy that can map its own format, load-balancing is possible across multiple instances of our service. We furthermore use simple crawling as a strategy to extract metadata (in particular, structural properties) about Web resources, in the same way as is described in [74]. Finally, our service can collaborate with other discovery services (including traditional Web search engines), in a process that we refer to as *deferred discovery* and that is enabled using OpenSearch descriptions in our case.

Microformats Microformats [256] are a very straightforward way of embedding semantic information directly in the representation of a Web resource by re-using HTML tags. In this way, they integrate information for both, people and machines, within the same document. Several search engines exploit Microformats when indexing Web sites, most prominently the **geo** and **adr** formats, both of which are part of the **hCard** Microformat that itself is a Web representation of the widely used vCard directory profile [290]. Another Microformat stands out as an interesting candidate for service integration on the Web: the **hRESTS** format [105] allows to expose information about the REST API of services that are provided by a Web resource. For instance, this can be used to describe how a specific service can be used by clients – in the example in Listing 6.4, the *ACME Hotels* service is invoked by sending a **GET** request to the endpoint at `http://example.com/h/{id}` where *id* is replaced by the appropriate hotel ID.

The main advantage of Microformats is that they enable developers without any experience with semantic technologies or ontologies to embed basic semantic metadata in their Web resources. A major disadvantage, though, is the lack of a way to automatically process many Microformats, as the included information is usually rather high-level. This is, for instance, the case with **hRESTS** which does not enable machine clients to deduce how a service can be used: clearly, the description shown in Listing 6.4 cannot be readily used by machine clients to deduce the service API, especially with respect to the *id* parameter. Another downside of Microformats is that they overload the HTML **class** tag, which makes it difficult for parsers to differentiate between semantic metadata and

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/">
3   <ShortName>Google</ShortName>
4   <Description>Perform a Google Search.</Description>
5   <Url type="text/html" template="http://google.com/search?btnI&amp;q={
     searchTerms}"/>
6 </OpenSearchDescription>
```

Listing 6.5: OpenSearch document that describes the Google search engine.

markup used solely for styling purposes. The novel *microformats2* approach, a second reincarnation of Microformats, represents an attempt to overcome this syntactic problem, but does also not enable more automatic processing of Web documents.

Microdata Microdata [330] is an HTML5 specification by the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG) that, similar to Microformats, aims at facilitating the embedding of semantic information within the HTML representation of Web resources. Other than the Microformats approach, Microdata uses distinct tag attributes to specify this information, which is beneficial because it facilitates the parsing of the resource representation. The Microdata format depends on vocabularies such as `data-vocabulary.org` or `schema.org` that specify the meaning of terms used in Microdata markup, such as *Person* or *Locality*. Although there are as of now no public efforts to standardize a description language for REST services based on Microdata, several search engines, including Google, Bing, and Yahoo!, use it to improve the indexing of Web pages and the presentation of search results – all three are part of the initiative behind the definition of the `http://schema.org` vocabulary and recommend to use Microdata for annotations (although they also still support Microformats and RDFa) [274].

RDFa The Resource Description Framework in Attributes (RDFa) [328] allows to embed structured data about a Web resource within its representation. Similar to Microdata, RDFa specifies only the syntax for embedding this data within Web resource representations and relies on independent vocabularies and taxonomies such as Dublin Core [259] or `schema.org`. We discuss the usage of semantic markup in greater detail in Chapter 7.

Deferred Discovery By means of *deferred discovery* strategies, our discovery service can make use of external services and integrate resource descriptions that are provided by these with results obtained from applying its own discovery strategies. To enable this, our software integrates a simple interface for the registration of such services that takes as input information about an external discovery service in the form of an OpenSearch document [247]. For instance, submitting the document in Listing 6.5 to this interface would register the Google search engine as an external discovery service. The deferred discovery functionality of our discovery service enables the creation of *discovery federations*, meaning that multiple discovery services can work together, thus enhancing interoperability and their ability to adapt to future description formats. Furthermore, this feature allows

for a very straightforward load-balancing mechanism: since instances of our discovery service themselves expose a description of their API in the OpenSearch format, multiple such entities can act as a single more powerful unit, by having overloaded instances relay queries to others. Finally, deferred discovery could also be used to alleviate the security issues regarding the injection of code that implements new discovery strategies: a third party could set up a local instance of the discovery service that includes the required strategy stub and register that instance as an external service with a central entity. This approach, however, has the downside of lowering the performance of the service federation as a whole due to the required remote invocations.

6.3.2.6 Service Discovery for an Infrastructure for Smart Things

In summary, we propose a system that enables the future-proof discovery of services that are provided by smart devices. Clients can use this service as a common interface to obtain information about smart things such as their name, location, and a human-readable description. We have also briefly described a method of embedding partially machine-readable information about the REST interface of a service in the form of the hRESTS Microformat – this possibility is explored in much greater detail in Chapter 7. A discovery system for smart things and their services is, however, not only useful in the context of the direct interaction of a user with a smart thing – rather, we motivated our discovery service by stating that a search infrastructure for smart things requires this functionality to be able to index devices and their services for look-up.

6.3.3 Smart Things Look-Up

After Web resources have been indexed using the discovery service, they are registered to one of the nodes of our management infrastructure and are thereby made searchable globally for clients of the system. The concrete node that registers a resource is determined from the information that the discovery service provides about that entity: a resource is always registered to the node that is responsible for administering the authoritative domain it is situated in (as determined by the discovery service), regardless of which of the infrastructure nodes receives the initial registration request. In this section, we discuss the look-up service of our infrastructure – specifically, we describe the different types of queries that clients can use to control the scope of a look-up operation, the routing of queries between nodes of the management infrastructure, and the local matching of smart things that are registered with a specific management node to a query.

6.3.3.1 Query Types

When initiating a look-up, clients can include information about the scope of their query together with the data that is used to select resources to be returned in the response from the look-up service. The system uses this additional information for two purposes: First, queries are routed to the management node best suited to answer them – for instance, a query for “a highly available temperature sensor in the building CNB” is routed to the administrative node of the CNB location. Second, the scoping information allows to pre-empt

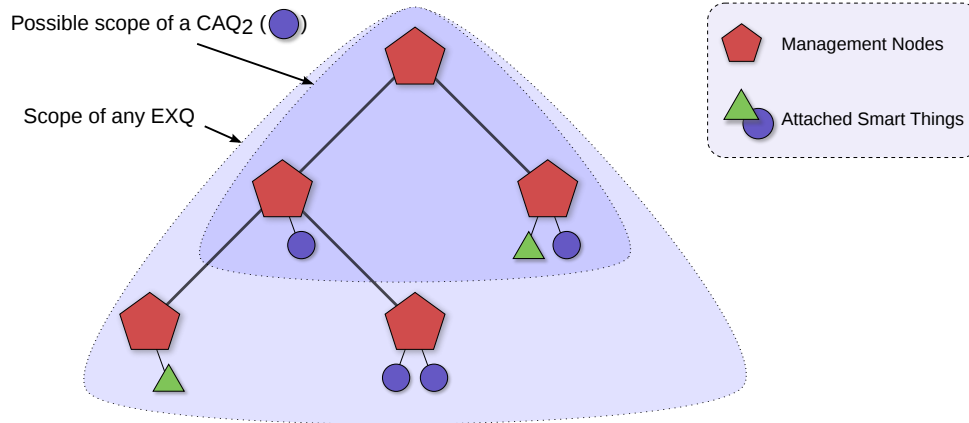


Figure 6.3: Query types: *EXQ* (EXhaustive Query) - search the entire subtree of a node. *CAQ_k* (CARDinality Query) - search for k corresponding resources within the subtree of a node.

queries once the required information has been found and thus increases the performance of the system – in the above example, the CNB node will relay the query to its sub-nodes that are responsible for managing the different floors of that building. If one of these nodes returns a response very quickly, the others need not further propagate the query to nodes that administer individual rooms, thereby conserving system resources. At the same time, this enables straightforward load-balancing as those parts of the management infrastructure that carry most load at any moment are on average slower to respond – subsequent messages of the client that triggered the look-up will consequently focus on the less strained part of the system.

To give clients control about the scope of their queries, our look-up service features three different query types: *Exhaustive Queries (EXQ)* consider the entire subtree of the queried node. *Cardinality Queries (CAQ_k)* are triggered to search for exactly k resources that match a query (see Fig. 6.3). Finally, to restrict a query to a specific subtree of a node, or to search at a wholly different location, the infrastructure uses a special type of query called *Request for Query (RFQ)*. RFQs are created internally whenever such a request is received – in essence, they are structures that hold one of the other two query types and, additionally, information about their destination: our infrastructure takes care of routing RFQs to the administrative node for their destination which unpacks them, executes the contained query, and transmits the answers to the initially queried node (see Fig. 6.4). The client remains unaware of the internal routing and receives its response from the management node that it initially connected to.

Internally, all queries are represented as data structures that contain a unique query ID, the URI of the node that initially triggered the query, the query type (including a positive integer number for CAQs), and information that is used to identify smart devices and services that match the query.

6.3.3.2 Internal Processing of Queries

Our infrastructure treats each query it handles as a Web resource, meaning that queries are identified using URIs and that queries themselves have a REST API: the clients of a query

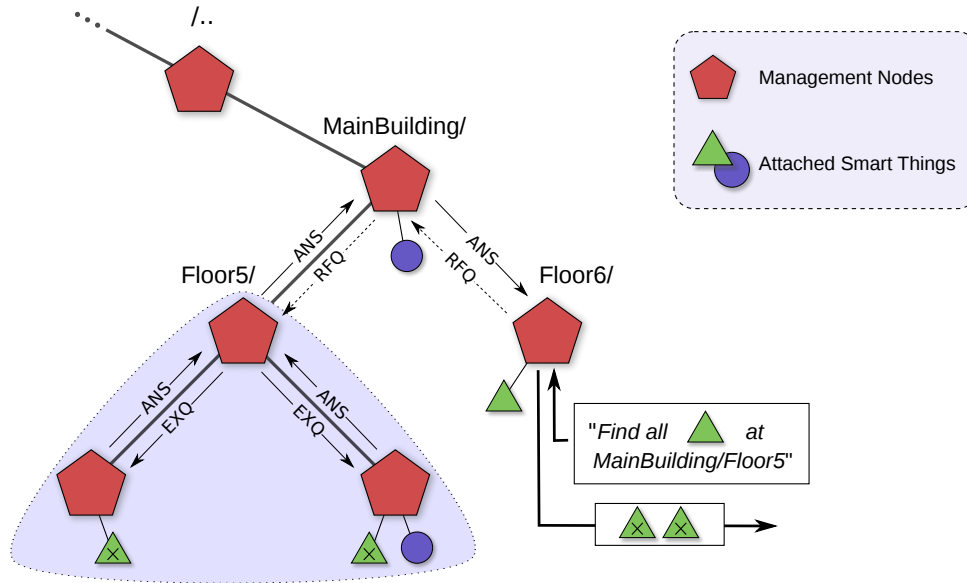


Figure 6.4: To trigger an *EXQ* at location `../MainBuilding/Floor5`, a client contacts its local management node at `../MainBuilding/Floor6`. Internally, the query is routed as an *RFQ*.

resource are management nodes that process that query, and the query resource itself is hosted by the management node that initially triggered the query. Indeed, whenever an infrastructure node receives a query from a client and is not able to satisfy it from its local database of registered smart things, it immediately creates a Web resource locally that corresponds to the query. Only after this resource has been created, the node relays the query to its children that each do a local look-up and post matching resources to the query resource (indeed, using an HTTP `POST` request). As soon as enough answers have been received at the query resource (e.g., as soon as two answers have been received for a *CAQ₂*), the management node prunes the query resource and relays the received responses to the client. This mechanism is useful to reduce answer times to queries, the number of messages induced by queries, and also has advantages with respect to load balancing. We discuss the impact of adopting this “resource-oriented view” on queries in more detail below, after describing the query routing process in more detail: we start with a discussion of *RFQ* handling and then move on to explain how *EXQs* and *CAQs* are processed – at multiple occasions in the text, we refer to Fig. 6.5 that illustrates the routing process on a more abstract level. Note that, in accordance with our design goals set forth in Section 6.3.1, the query processing does not depend on global knowledge about the management infrastructure.

Routing of RFQs From a processing perspective, the simplest type of query is the *RFQ*. By examining the destination location parameter of an *RFQ* (node (b) in Fig. 6.5), the node that receives such a query can determine whether it should unpack the *RFQ* and trigger the contained query locally (c) or whether the *RFQ* should be relayed to one of its sub-nodes or its parent (d). When routing the query to another node within its subtree, it uses the destination information to find out which of its child nodes is best qualified

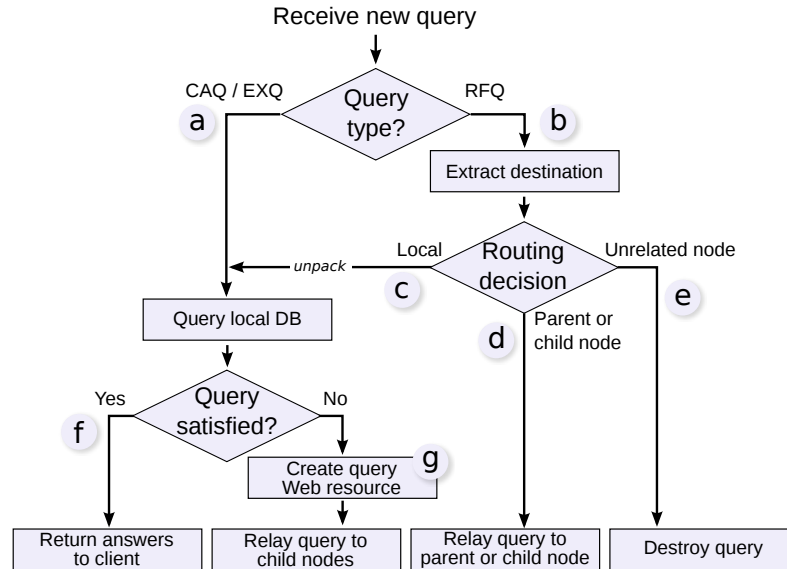


Figure 6.5: Overview of the processing of received queries by a node of our management infrastructure.

to process it. If it determines that the destination node is located in a different sub-tree altogether, or that it is a (transitive) parent, it routes the query to its parent node. If the destination location is unrelated to the location of the current node (i.e., the two nodes have no common ancestor), the query is destroyed (e).

Routing of CAQs and EXQs Upon receiving a query of one of the other two types, a management node first checks whether it can serve the query from its local database (node (a) in Fig. 6.5). If this is possible, the responses to the query (i.e., matching locally managed resources) are delivered to the client (f). Else, the management node starts to collect answers from other management nodes in its sub-tree until the query is satisfied at which point the answers are, again, delivered to the client. As discussed above, to collect answers from its sub-tree, a node first creates a Web resource that represents the query locally (g). Next, it relays the query to its direct children that also do a local look-up and post obtained results to the query Web resource. Each node that receives a query follows the same procedure, with one exception: only a single Web resource is created per query, i.e., management nodes that receive a query from another node in the infrastructure skip step (g) of the query processing algorithm. At any time, nodes can find out whether a query has already been satisfied and, consequently, should not be further propagated within the infrastructure, by requesting that information from the query Web resource.

This mechanism of handling queries via explicit Web resources allows to decouple management nodes in the look-up process: nodes that trigger a query in their subtree need not wait until they have obtained all answers from their sub-nodes in a way similar to the propagation of queries in a wave algorithm. Furthermore, due to the design of the mechanism, answers to a query are automatically sorted according to the response times of the management nodes that give them – thus, resources registered to already overloaded management nodes are on average returned less frequently than those from nodes with more capacity to spare. Indeed, because management nodes act as “gateways”

to the sub-tree they administer, the system automatically treats sub-trees in a way that considers the load they are bearing at the moment. Note that the load balancing does, however, not influence the correctness of the look-up system: a response that is returned from a management node which bears less load is as correct an answer to a specific query as one from an overloaded one. For instance, if a client submits a query such as “Three temperature sensors on the 5th floor of building *MainBuilding*,” this implies that it is not interested in where exactly those temperature sensors are located on the 5th floor.⁷ On the contrary, this system promotes answers from nodes higher up in the hierarchy that have a more general scope: if the node responsible for *MainBuilding/Floor5* is capable of fully servicing the query from its local database, it does not need to contact other management nodes that administer specific rooms.

The proposed mechanism furthermore helps to improve the performance of look-up operations within our infrastructure in several different ways: The response time to client CAQs is lower because the triggering node does not have to wait for all nodes to deliver their answers. Rather, as soon as the local query Web resource indicates that the query has been satisfied – i.e., if k answers have been received for a CAQ_k – it can destroy the query resource and send the answers to the client. This in turn allows to pre-empt queries that are still being processed by other nodes in the system, as other management nodes can use the query Web resource to determine the current state of a query: if the query resource is gone (as indicated using an appropriate HTTP response code), all nodes that are still processing the query can safely destroy it and will not propagate it further. Additionally, when a management node posts locally matching resources to a query resource, it receives as response the number of answers that are still required to satisfy the query. When that number reaches zero, the query can be destroyed as well.

Processing of EXQs Because it allows for queries being pre-empted, the implemented mechanism is very beneficial when routing *CAQs*. For *EXQs*, however, the triggering node is not aware of how many answers it should expect at the query Web resource. Therefore, for this type of query, the look-up service includes an explicit notification mechanism regarding pending answers from nodes in the sub-tree of the triggering node: before nodes in that tree propagate a query to their child nodes, they register this child as pending at the query Web resource. As soon as the child node delivers its answers, this flag is cleared again – thus, the triggering node knows that it has received all answers as soon as there are no more pending flags. The same mechanism avoids stalling when a CAQ_k is triggered that cannot be fully satisfied because the number of matching resources in the system is smaller than k .

Advanced Query Caching Finally, our resource-based query routing mechanism allows for smarter handling of queries that are similar to others which are already being processed by the system. If a management node receives a query that is similar (i.e., equivalent or overlapping) to another query that already has a corresponding Web resource and is, thus,

⁷If the client was interested in obtaining sensors specifically in the office “502,” it should include that information in the query!

MainBuilding/Floor5 that match the keywords *Temperature* and *Sensor*, a client would enter this line:

3x Temperature Sensor at MainBuilding/Floor5

To trigger an EXQ that finds all resources that match the keywords, the client would use the “All”-keyword (i.e., enter “*All Temperature Sensor at MainBuilding/Floor5*”) and the interface allows to use the wildcard character (*) instead of specifying keywords inside a query string (i.e., “*All * at MainBuilding/Floor5*” is a valid query for any resource at that location). Finally, clients can choose to omit the cardinality and/or location specifiers: if both are left out (i.e., “*Temperature Sensor*”), the node triggers a local CAQ_1 . For the internal keyword-based look-up within the local database of a management node, we use the Sørensen-Dice coefficient [47] to compare the keywords to information about the resources that are registered with the node such as their name, identifier, description, category, brand, or reviews. Adapted for information retrieval in text documents, this measure takes the ratio of character bigrams that are equivalent in the keyword string and the resource information to the total number of bigrams for determining the degree of similarity, i.e., how well the resource information matches the specified keywords. In our matcher, different types of information about the resources are weighed differently: for instance, the similarity between a keyword and the unique identifier of a resource, or its name, carries a bigger weight than the measure of agreement between the keyword and a review about the resource. Adopting Dice’s coefficient for string matching has the advantage of being robust to typographical mistakes and produced good results when compared to other algorithms such as the edit distance or longest common substring.

Machine Clients For machine clients to find resources within our management infrastructure, we have implemented another interface that is better suited for machines by enabling them to use more structured queries to describe the resource they are attempting to locate. Technically, machines use the same API as people do. However, other than human users, they submit queries not as HTML forms but in the JSON format. This gives them the liberty of tuning queries much more than humans can – for instance, machines can submit queries for resources that have a “rating higher than 4.5” or whose REST API “provides the service *ACME Hotels*” (see Listing 6.4 on page 98). The same interface also allows to specify the destination and scope of a query.

6.4 Deployment and Evaluation

Our management infrastructure was deployed to administer six rooms in the office space of our research group, over the period of 11 months in the years 2011 and 2012. Additionally to these embodied nodes that were running on Norhtec MicroClient Sr. devices, the structure contained six virtual nodes that were responsible for administering its hierarchically higher levels (two floors and our office building) as well as three more rooms without any physically deployed devices. Each of the six rooms that were directly covered by a node contained a SunSPOT sensor node, three rooms additionally contained

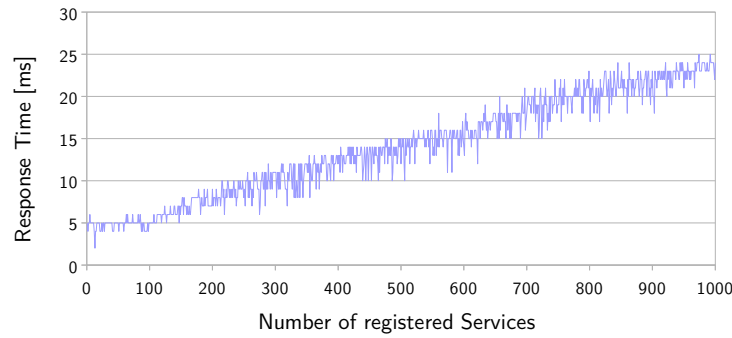


Figure 6.7: Deployment of up to 1000 services on a single node: the chart shows the median response time to 100 *EXQs* that return up to 1000 results each.

a Ploggs smart meter, and one of them contained several other smart things such as a Web-enabled RFID reader, a toy robot, and a Web-connected loudspeaker system. Additionally, virtual SunSPOTs and other purely virtual devices (thermometers, electricity meters, etc.) were deployed at the nine room-level management nodes. The infrastructure administered these devices and proved resilient to changes in its structural configuration as well as nodes failing and recovering again. However, due to the limited availability of actual connected smart devices, we were not able to test its scalability with respect to increasing the number of devices to administer. Extending the infrastructure to include more *management nodes* increases the size of its administrative domain but does not strain the system, since the larger coverage area is divided into sub-domains that are each taken care of by a node. Given the design of the infrastructure, where global communication is unnecessary except when triggered by explicit client requests, we can claim that our approach allows the system to expand to large domains without scalability problems, given that the number of resources *per administrative domain* remains constant.

6.4.1 Simulation Environment

To simulate the smart devices that are registered to individual nodes in the infrastructure, we used a custom-developed simulation environment for Web-enabled devices. In this environment, Web resources can be created that simulate specific types of sensors and actuators. The representations of the simulated resources (in the HTML, JSON, XML, and plain text formats) and their behavior is defined using resource templates that allow users to, for instance, specify the resources' response behavior reliability and response time. Sensors can furthermore be configured to deliver values according to a statistical model and can also be set to transparently fetch their result from remote sources, enabling them to act as proxies for physical devices. Using this simulation environment, we tested the deployment of up to 1000 resources to individual management nodes which did not cause any major performance issues in the discovery and look-up components (see Fig. 6.7). Thus, because the infrastructure is able to administer high numbers of nodes within the domain of a single management node and the structure as a whole can expand to include many such nodes, we claim that our approach of segmenting the complete administrative domain of the infrastructure can support a large total number of devices.

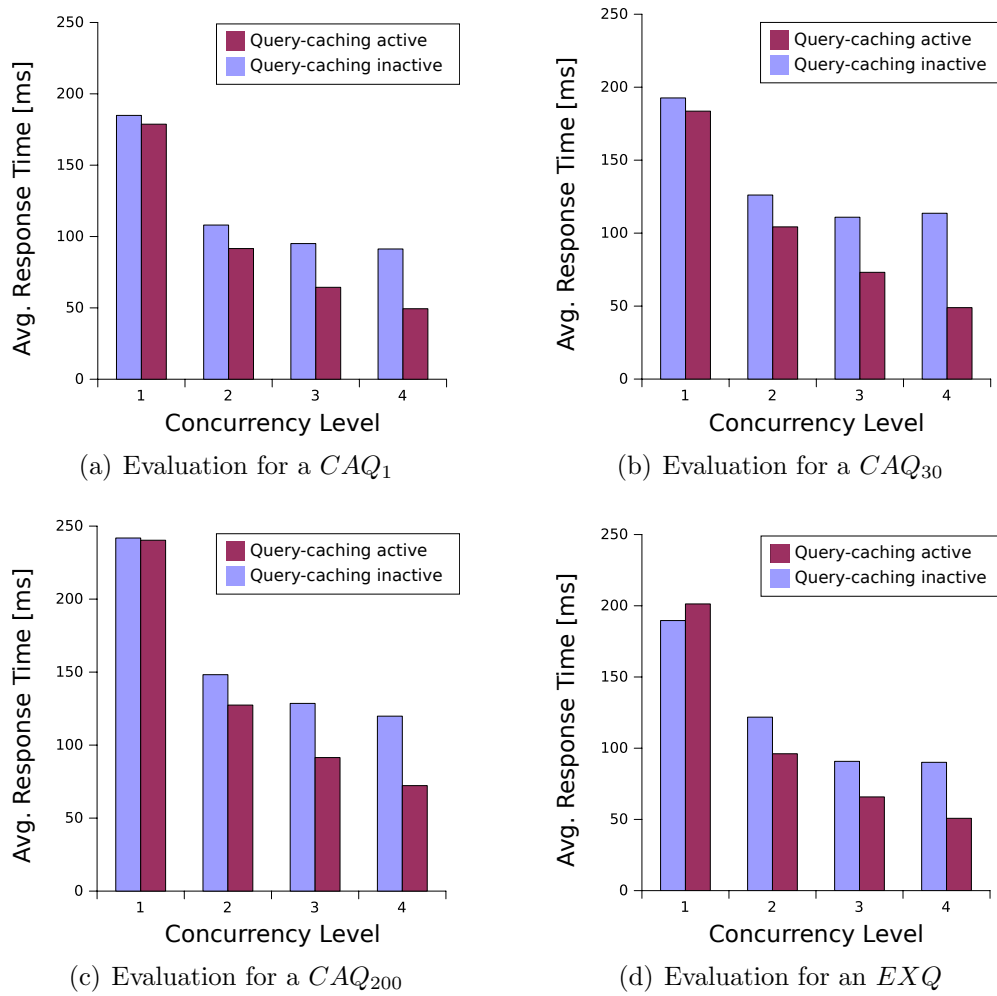


Figure 6.8: Simulation results for comparison of active vs. inactive query-caching for different types of queries and concurrency levels.

6.4.1.1 Query Caching

To additionally evaluate the performance of the look-up mechanism of our infrastructure in a more extensive setting, we deployed nine management nodes (one on building level, two on floor level, six on room level, see Fig. 6.1 on page 91) and registered a total of 600 simulated sensors and actuators (temperature sensors, electricity meters, light switches, etc.) uniformly at random with the room-level nodes. For our evaluations, the entire infrastructure was deployed on a single machine that was running the individual management nodes in parallel.

We evaluated the performance impact of the infrastructure’s query caching mechanism using the *apachebench* tool [248] with four different types of queries (a CAQ_1 , a CAQ_{30} , a CAQ_{200} , and an EXQ). For each of these queries types, 1000 requests were issued on different levels of concurrency. Our expectation was that our system should increasingly outperform a baseline system that does not treat queries as resources and therefore does not provide the described advanced caching as the concurrency level of the queries increases. Indeed, we observed this behavior (Fig. 6.8): for CAQ_{30} requests and

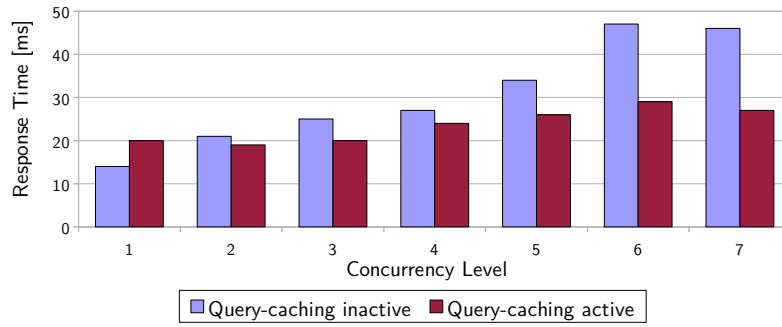


Figure 6.9: Response times to $EXQs$ that are sent to a deep hierarchy of six management nodes with ten registered services each. The query-caching system increasingly outperforms the baseline system for higher request concurrency levels. Similar to Fig. 6.8(d), the overhead of this system is visible when requests are received strictly sequentially.

at a concurrency level of four, the response time of the caching system is approximately half that of the baseline system. For all four query types, the caching system is faster at higher concurrency levels, but the differences vary across the query types.

A peculiarity that we can observe in the data is that $EXQs$ perform remarkably fast compared to the other three query types. The explanation for this behavior is given by our optimizations for EXQ queries described above, i.e., that these queries are forwarded immediately to all children of a management node and the local node does not consider preempting them. This advantage vanishes when the management nodes are not deployed on the same machine (and, thus, network overhead becomes relevant), and for queries that are triggered in deeper hierarchies: here, the other query types can be preempted at higher levels in the hierarchy, thus accelerating their processing. For the same reason, the effect is a lot less pronounced when management nodes at intermediary levels of the hierarchy hold more resources which enables the preemption of queries in the first place – in fact, in the setup that we used for the evaluation at hand, only nodes at the room level contain information about registered resources and, thus, every single query is propagated to the very bottom of the hierarchy with no preemption events can occur.

Our test also shows that queries of type CAQ_1 are answered only slightly faster than queries of type CAQ_{30} . This is because, on average, about 100 resources are registered to each node at room level. Therefore, the difference in response times between a CAQ_{30} and a CAQ_1 only amounts to the difference in the retrieval of the elements from the local database of the management nodes and the passing of information about these resources to the client: clearly, the response to a CAQ_{30} is considerably larger than that to a CAQ_1 . Finally, we did another analysis to investigate the effect of the query caching in more detail, and in complete isolation from the query preemption: Fig. 6.9 shows response times for an EXQ (which cannot be preempted) on a deep hierarchy of six management nodes (i.e., we assigned the following place identifiers to the deployed nodes: a , a/b , ..., $a/b/c/d/e/f$) with ten registered services per node. Also in this setting, our system increasingly outperforms the baseline system when the concurrency level increases.

In summary, our simulation results demonstrate that the query caching is valuable for achieving higher querying performance in the proposed management infrastructure, given

that there are enough similar queries (which was triggered by higher concurrency levels in our experiments). When deploying the management nodes in a distributed way and thus also incurring latencies due to the network communication, we expect that the difference between a caching system and the baseline system will be even greater because, in this case, the caching and preemption of queries can avoid network overhead as well.

6.4.1.2 Load Balancing

We have already hinted that the look-up mechanism of our infrastructure automatically takes care of rebalancing load from overloaded nodes to idle ones. This property was also visible in our tests: due to the lexical ordering of child nodes in the local database, the infrastructure usually (i.e., in more than 90% of the cases) delivered a resource from the *Floor5* branch when a CAQ_1 query with the keyword “temperature” was triggered at location *MainBuilding*. To test the load balancing, we flooded the management node responsible for administering Floor 5 – as soon as this started, the same CAQ_1 as before returned resources in the *Floor6* branch in more than 96% of the cases. Because the scope of our CAQ_1 was *MainBuilding*, all answers that include a resource from either rooms at Floor 5 or Floor 6 are equally acceptable.

6.4.2 Query Visualization

As we already mentioned in Section 5.4 in the previous chapter, our visualization tool for device interactions in smart environments can also be applied to visualize distributed algorithms whose execution involves multiple endpoints. Figs. 6.10(a) and 6.10(b) show the graph-based interface of that tool when monitoring nine nodes of our management infrastructure in two different setups while executing look-up operations on behalf of a client. In particular, this illustrates that our look-up service creates local query resources that are updated with responses from other nodes within the infrastructure (in Fig. 6.10(a), one of the HTTP POST request to the query resource is highlighted). For comparison, Figs. 6.10(c) and 6.10(d) display the same requests when triggered on the baseline system that was also used in the evaluations discussed above – this implementation uses a wave algorithm to collect information about registered resources from all nodes in the infrastructure.

6.5 Summary

In this chapter, we described our implementation of a distributed management infrastructure for smart devices that supports people and machines when searching for smart things and the services they provide. The infrastructure consists of a hierarchy of individual management nodes each of which is responsible for managing a specific location such as a room, floor, or building. The hierarchical structuring of the infrastructure is beneficial because it subdivides its global administrative domain into smaller, widely decoupled cells, thus guaranteeing the scalability of the system. Furthermore, it allows to exploit the locality of device interactions in smart environments. Two services form the cornerstones

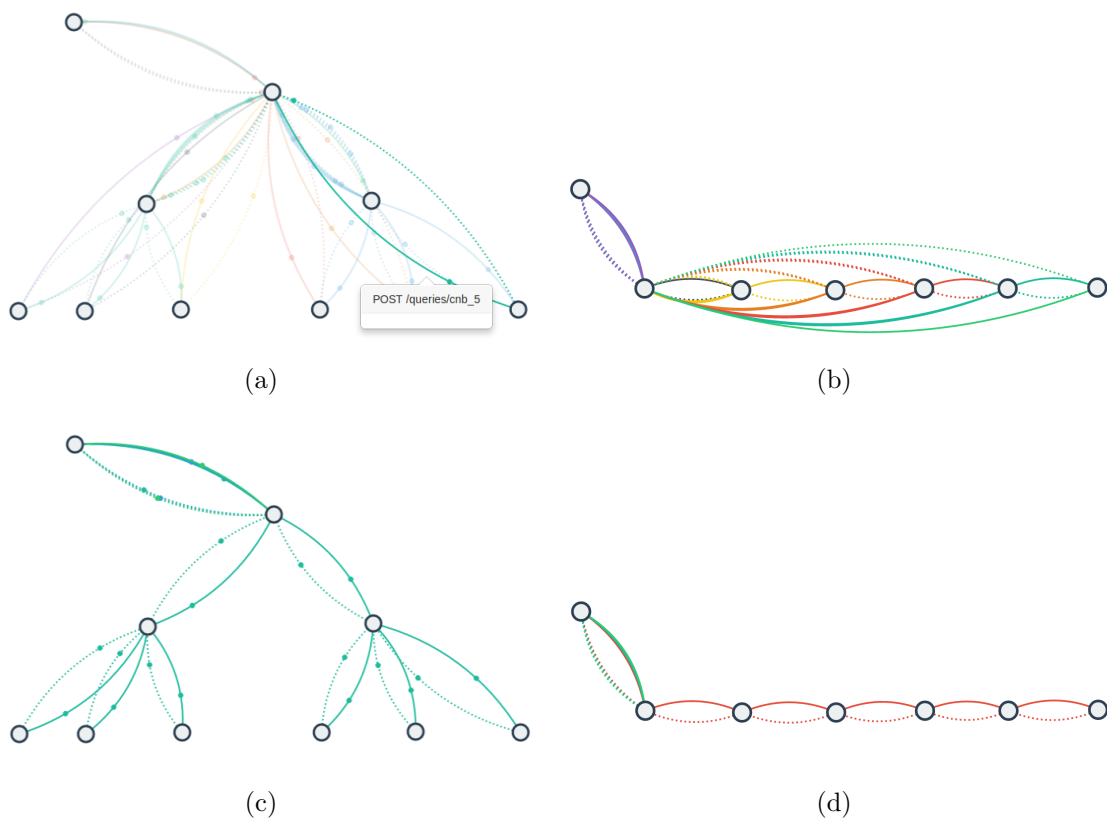


Figure 6.10: Application of our tool from Chapter 5 to visualize interactions between nodes of our infrastructure: (a) and (b) show look-up operations by a client (top left node) for the node hierarchy shown in Fig. 6.1 on page 91 and a deeper node hierarchy. In both cases, the nodes send their answers directly to the initial node. (c) and (d) show the same requests on a baseline system that executes a simple wave algorithm to find matching resources.

of our infrastructure: an *extensible discovery service* enables it find metadata about Web services and parse it to a common internal format for indexing which is based on the *Smart Things Metadata* model that was suggested in [73]. A *look-up service* enables clients to search for registered and indexed services using both a keyword-based interface and one that is based on more structured information about the kind of resources to be found.

Our infrastructure enables people to locate devices and services locally in their smart environment and globally within the scope of the entire system. It also allows machines to query for services that provide a specific functionality, as specified by metadata about their REST API. However, due to limitations of the formats that are used to describe the functionality of Web services (for instance, when using the *hRESTS* Microformat), machines are at this point merely able to find services that roughly match the desired characteristics, but cannot make use of their functionality because they are unable to invoke them – to do this, they would require additional machine-readable information about the service API. Furthermore, the information that our infrastructure uses to index services only gives a rough impression of what functionality they provide. While this kind of information is sufficient for people to understand what a service does, a machine client cannot use it to infer its functionality. Enabling machines to process this kind of information is the main topic of the next chapter.

CHAPTER 7

Service Composition in the Web of Things *

After discussing how smart things and their services can be described and searched for based on structured metadata that they provide via their Web interface in Chapter 6, the aim of this chapter is to lay the foundations for enabling machines to use services that are provided by devices in smart environments *automatically* and to *combine them to achieve higher-level functions*. This will enable machines to automatically create and invoke “physical mashups” that integrate functionality provided by multiple individual smart objects and that can also include capabilities of traditional Web services. Such composite applications on top of individual services in smart environments are expected to become important in many application domains that stretch from industrial automation to individual well-being at home [91]. For instance, in the smart manufacturing domain, individual machines could automatically cooperate to create a final product [216]. In the medical domain, readings from personal heartbeat sensors could automatically be processed and an ambulance could be dispatched when cardiac arrhythmias are detected. At home or in hotel rooms, ambient intelligence systems could make sure that the current configuration of the environment matches the inhabitants’ preferences, for instance with respect to temperature, lighting, and ambient music – in hospitals, the same system could control the local oxygen saturation to aid asthmatics.

In this chapter, we propose two mechanisms that we developed to enable the automatic collaboration of services in smart environments. The first of these addresses service integration challenges on the syntactic level and enables the semi-automatic composition of services – in essence, we have developed a Web-based *computational marketplace* where mashup developers can publish information about service compositions that can then automatically be reused by machines. The second mechanism that we describe is a fully automatic service composition system that uses functional semantic metadata to describe a service’s functionality and its API in a machine-readable way. This system enables machine clients to reach specified user goals by inferring service compositions at runtime using a semantic reasoner.

In the following, we first present several approaches that were employed by others

*This chapter is based on the following published articles: [130, 131, 132, 138, 139, 140]

to enable automatic service composition as well as systems that aim to facilitate the composition of services in smart environments for end users. We also discuss the role of the HTTP HATEOAS constraint when composing REST services. Next, we present the first of the above-mentioned two approaches in Section 7.2 along with an evaluation of this system in a healthcare scenario. We continue with a discussion of the potential of semantic metadata and reasoning to enable the machine-readable description of service capabilities and their APIs, and, in Section 7.3, propose a concept that allows machines to automatically derive device mashups in smart environments. We present benefits and drawbacks of our system, and in particular discuss how the added complexity that is due to the employed semantic metadata can be made manageable for end users in Section 7.4.

7.1 Service Composition for Smart Things

One main goal of the Internet of Things is to empower users by giving them the ability to “program” everyday things and create new public and personal services on top of IoT-integrated devices. Given a reliable infrastructure that supports the semantic discovery and look-up of smart devices and their services, it is thus required that users can easily learn *how to use* the discovered services (e.g., fetch data from sensors and trigger actuators) and that they can *combine* the capabilities of different devices and services to create advanced functions that provide added value. Due to the adoption of Web patterns for the provisioning of services by smart things in the Web of Things, using their functionality is simple for users (this literally is as simple as browsing the Web), and also machines can automatically deduce how a service can be *invoked* because the protocol semantics are specified by REST. However, enabling users to *combine* Web services remains a big and heavily researched problem [29, 91] – in fact, it is one of the central challenges in the domain of *End-user Programming*.

Providing users with the tools to program their smart environment has been identified as an important challenge for ubiquitous computing research for instance in the smart home domain [29]. Here, many heterogeneous devices that can modify the environment (e.g., smart thermostats) or provide contextual information about the user’s home (e.g., motion sensors) could interact to enable more complex applications that involve multiple cooperating services: for instance, motion sensors and smart thermostats could together infer and apply an optimal heating schedule for the home [102]. The same is true for smart factories, where the easy (re)configuration of manufacturing environments is gaining importance [111, 216] and operators must be supported in managing the implied added dynamicity. The combination of smart things services is not only beneficial when considering actuation but also when processing data from physical sensing devices: to realize the vision of a “smart world” that helps users make faster and more well-informed decisions, it is required that raw sensor data can be obtained, processed, and published openly in a consistent way.¹ Thus, while the WoT gives us access to unprecedented amounts of raw, real-time data, it does not imply a clear method of how services could interact with this data and with each other to consistently derive higher-level information from it.

¹Note that by “open” we are not referring to unsecured information but to openly accessible data.

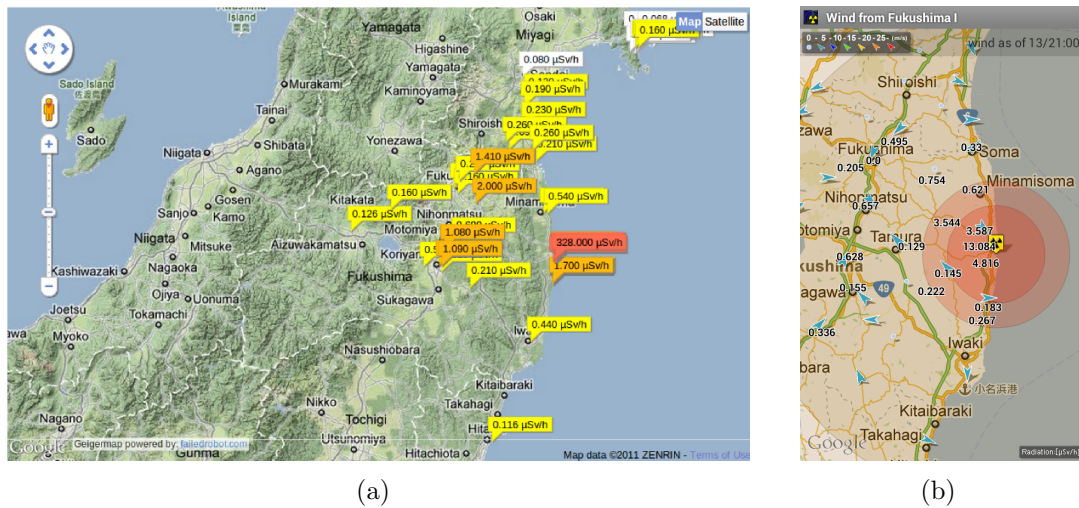


Figure 7.1: (a) The “At a Glance” application displays real-time radiation information from publicly available data feeds from Geiger counters. (b) The “Wind from Fukushima” application mashes Geiger counter readings with information about wind conditions.

To illustrate how open models can enable the interaction of distributed services to process raw sensor data and integrate public sensor streams, we review an intriguing community movement that aimed at providing the general public with vitally important information in the onset of a series of meltdowns in the Fukushima nuclear power plant in the year 2011: shortly after the incident, ordinary citizens started to deploy Geiger counters and published their data to an online data aggregation platform called Pachube [258].² Others developed applications that used this raw data to derive higher-level knowledge and published their results on the Android marketplace, wrapped in applications for the Android operating system: an early example of such an application is “At a Glance” [337] which visualizes radiation data in real time (see Fig. 7.1(a)). Other applications integrate several data sources to provide more refined information: “Wind from Fukushima” [297] combines radiation readings with information about the current wind conditions and the user’s location to issue location-based radiation warnings (see Fig. 7.1(b)).

In this example, the open publishing of raw data was crucial to enable the crowd-sourced processing of radiation data and the resulting impressive applications to make that data accessible for the public. In principle, the same benefits could have been delivered by a company that would have deployed the sensors, collected their data, processed it, created applications for users to consume it, and distributed these applications. However, this is not a very realistic scenario for multiple reasons: developing the entire end-to-end system requires a very broad range of expertise, thus deterring a single company from carrying the effort alone. Furthermore, a company would first need to develop a viable business case for the system, thus delaying its implementation or dismissing the project altogether. On the other hand, making the collected data accessible to a large community of developers allows everyone to apply their own know-how for analysis and visualization, supported by an open and widely used platform such as the Android marketplace.

²After several takeovers, this platform is today known as “Xively” [301].

We argue that this observation can be applied to many other domains as well, for instance in the context of smart homes: a home automation system that is sold by one company can often only integrate appliances sold by the same enterprise and is not interoperable with automation systems from different companies that could be running next door. To enable applications that use information from many smart homes in a neighborhood – for instance, to achieve broader demand-side electricity management – would require all households to install the same automation system. Even if that was possible, this neighborhood would perhaps be unable to collaborate with others. In a paradigm that is based on closed systems, as illustrated with this example, it is thus necessary that all devices, as well as the infrastructure between them, are delivered by a single company or a consortium that uses interoperable technologies.

According to [207], over 28000 Web services are currently openly available on the Web – these are advertised via platforms such as *WebServiceList*, *WSIndex*, or *ProgrammableWeb* [318] which is one of the most prominent directories of Web APIs: *ProgrammableWeb* currently hosts information about more than 5000 APIs (e.g., text analysis services, restaurant reviews, etc.) and more than 6000 manually created mashups that provide aggregation, analysis, and visualization services on top of these APIs. Additionally, the platform provides capabilities for developers to comment and rate APIs and mashups in an effort to foster a community around the creation, provisioning, and usage of Web-driven data processing tools – platforms such as *ProgrammableWeb* make it obvious that developers are not only willing to develop remote aggregate computations and publish them but that many also wish to expose the *results of their computations* for use by third parties. Enabling the publishing, storage, and management of distributed data streams is the main task of another group of services that include *Xively* [301], *Nimbits* [309], *ThingSpeak* [296], *Evrythng* [271], *Paraimpu* [179], and *WoTKit* [21]. In the WoT, they are complemented by infrastructures for smart things which make devices and services discoverable and searchable for people and machines: their purpose is to give clients access to individual services, a prerequisite for enabling users to create advanced mashups on top of smart devices.

7.1.1 Approaches to Service Composition

We now take a step back and consider the challenge of creating aggregate services outside the WoT context: in the following, we review several approaches to the semi-automatic and automatic composition of general Web services, most of which are based on the WS-* paradigm. In particular, we discuss the drawbacks of composition systems that are proposed in the literature and highlight where the REST principles could offer a remedy. After this review, we propose two service composition systems that exploit REST for facilitating the composition of Web services and use semantic technologies and reasoning in conjunction with RESTful systems as the basis of a fully automatic composition system.

We consider service compositions in the form of *Service Orchestration*, meaning that the interaction between different devices and their services is coordinated by a central instance [177] – in our case, this is the client who aims at composing services to reach

its application goal.³ Such a goal could, for instance, be to configure the client's smart environment in a specific way, or to produce a specific product in a smart manufacturing environment. In the terminology of the *Web Services Business Process Execution Language* (WS-BPEL, or BPEL in short) [313], the de facto industry standard for describing service orchestrations, we thus view a user interaction with his smart environment as a "business process" that is executed on behalf of the user by a client application and involves a number of elementary Web services. On this abstract level, this view matches the REST architectural style which dictates that the control about the next steps in a Web interaction must lie with the client application, and that the server (i.e., the smart thing or Web service that the client interacts with) guides the interaction by advertising appropriate information about state transitions (see Chapter 2).

Current approaches to service composition by end users have also assumed this perspective – consequently, many adopt a *process-driven* service composition paradigm, meaning that users create a composite service by connecting multiple individual, elementary, services, for instance using a formal language or graph-based tools. This is true for many well-known systems in industry (e.g., IBM Business Process Manager, Oracle BPEL Process Manager, Microsoft BizTalk, and SAP NetWeaver; all of these systems provide support for BPEL), open source solutions that address business process modeling and execution (e.g., Apache ODE and JBoss jBPM), and research prototypes (e.g., WebDG [147], SOA4ALL [112], and eFlow [32]).

In contrast to the process-driven composition of services, *goal-driven* service composition aims at reducing the complexity of the development process as a whole by automating the composition step: instead of creating mashups by hand using scripting languages or graphical tools, users only model their goals, for instance by describing the desired state after a service mashup has been executed. Then, a composition tool is used to link individual services together and create the actual composite application that can be invoked by the user. Aside from the greater level of automation that can be achieved in goal-driven systems, they have another major advantage over the process-driven creation of mashups, because they enable the on-the-fly inference of the service mashup and thus avoid the static linking of services. Therefore, mashups created in this way are much more flexible than composite applications that follow fixed execution paths: they can automatically adapt to changes in the environment by incorporating services that newly appear and bypassing those that suffer outages. This makes service mashups fault-tolerant because it retains the potential to reach the user-specified goal even if services become unavailable. This characteristic is especially valuable in pervasive computing scenarios and with respect to mobile applications that face highly dynamic smart environments [33, 207].

7.1.2 Current Service Composition Methods

Many different approaches to service composition have been proposed in the last 30 years, a development that started even before the long-time standard set of Web service technologies (SOAP + WSDL + UDDI) was conceived around the year 2000. Today, and already

³An alternative service composition paradigm has been termed *Service Choreographies*. There, the emphasis is on the *distributed control* of interactions between different parties [177, 207].

for some time, “connecting to customers, suppliers, or partners electronically” is considered the top global management issue in the IT domain [177], thus necessitating tools that allow to compose (Web) services globally and across company boundaries. It quickly became obvious that the manual composition of services, where designers use a language such as BPEL *directly* to define a service mashup, is too time-consuming, inflexible, and error-prone, especially when considering the size of the Web and its dynamicity [207]. This led to the development of a great number of semi-automatic composition systems [152] that provide tools – often process-driven, graphics-based composition engines – to support the design process. Furthermore, since around the year 2000, fully automatic process composition engines have been proposed [207]. These typically take as input a set of descriptions of elementary services and a design goal and attempt to synthesize a composite service from the individual descriptions, using syntactic or semantic matching techniques that are often based on mechanisms known from the Semantic Web [18] or on (often graph-based) planning techniques.

Many of the semi-automatic and automatic composition methods express service mashups in one of several specifications that are in widespread use especially in the context of industrial business processes. The most prominent of these is WS-BPEL that was proposed by BEA Systems, IBM, Microsoft, SAP, and Siebel Systems and standardized by the OASIS consortium in the year 2007 [207]. A member of the WS-* specifications family, BPEL can specify business processes that depend on WSDL-described services using several different types of primitive constructs (e.g., **sequence** or **while**). BPEL does not feature a standard graphical notation, but many tools are available that propose their own notations to allow graphically modeling BPEL processes, such as the BPEL Designer Project for Eclipse [254]. Furthermore, several tools for the semi-automatic development of BPEL documents exist [34, 37, 157, 241] and some automatic service composition tools can synthesize composite services given an abstract BPEL specification and a business goal [207, 224]. Basic partial mappings between BPEL and the widely used Business Process Model and Notation (BPMN) have been proposed [237] – BPMN, in turn, has been extended with modeling capabilities for sensors and other smart devices [64, 151] and with *push*-functionality for REST business processes [175].

7.1.2.1 Drawbacks of WS-BPEL

With respect to *automating* the composition of Web services, the main drawback of BPEL is its reliance on WSDL as description language for the individual components of a service mashup: WSDL only specifies the low-level functions of a service and does not include a definition of its high-level domain semantics which is crucial when automatically creating meaningful composite applications [207, 231]. Furthermore, WSDL suffers from other drawbacks that make it unsuitable for the description of more dynamic REST services for purposes of ad-hoc tactical integration that becomes relevant in pervasive scenarios. For instance, past experience has shown that client stubs are frequently compiled from the WSDL files provided by servers (i.e., “top-down,” contract-first development) – this leads to a tight coupling of the two endpoints and is problematic because it hinders their independent evolution: when the server interface changes, the client breaks. For

the same reason, we are skeptical about current proposals that aim at repairing WSDL's shortcomings and making it applicable to resource-constrained devices, such as the *Devices Profile for Web Services (DPWS)*. Although this is being supported by several European research initiatives (e.g., [267, 269, 298]), it already suffers from strongly declining interest, which is also due to DPWS sticking to SOAP message envelopes that introduce high messaging overhead [81]. To enable semantic annotations within WSDL, the W3C has proposed *Semantic Annotations for WSDL (SAWSDL)* [326] which allows to link WSDL elements such as inputs, outputs, and operations to concepts in a semantic model – however, this specification has failed to attract much attention beyond academic examples because it inherits many of WSDL's shortfalls, primarily its brittleness and verbosity [231]. Still, in an attempt to make WSDL-style specifications applicable to REST interfaces, the *Web Application Description Language (WADL)* has been proposed. Similarly to WSDL, WADL does not include semantic descriptions of service functionality [231] and has been criticized for the same kind of contract-first-susceptibility that makes WSDL undesirable to describe more dynamic services [279].

7.1.2.2 Semantic Technologies for Service Descriptions

Moving toward the usage of semantic technologies for describing services, several description formats for REST services have been created that are based on so-called *poshformats* such as Microformats (e.g., *hRESTS*, which we discussed also in Chapter 6, and the *SA-REST* format [208]). While these include information about the API of a service and its basic functionality, they cannot automatically be used by machines to infer how to use a service, or how to compose multiple services that are described in this way (see Section 6.3.2.5 in the previous chapter). A different approach to service composition that puts semantics first and is based on the Ontology Web Language (OWL) [333] is OWL-S [302], an ontology for describing Web services that makes use of the Resource Description Framework (RDF) [334] model. An OWL-S description consists of three parts, one of which is the *service profile* that includes information about the service's functional properties (i.e., inputs, outputs, and preconditions) and non-functional characteristics (e.g., Quality of Service (QoS) parameters) [207]. While the precondition/postcondition style of OWL-S allows to describe the high-level functionality of a service, the description format is criticized for not expressing these conditions within the RDF document that contains the OWL-S description [231]. This means that languages such as the Knowledge Interchange Format (KIF) [65] must be used to connect them to the OWL-S document which represents a drawback when the documents have to be parsed and interpreted, and is probably one of the main reasons for the very limited uptake of the OWL-S mechanism [231] and the strongly decreasing interest in the format over the last ten years.

Apart from these languages that are prominent mostly in academia for the purpose of creating composite applications, industrial outfits rely on frameworks such as the *Service Component Architecture (SCA)* [311] and other standards such as the *Enterprise Mashup Markup Language (EMML)* [207]. Some of these systems have been successfully used within the context of semi-automatically creating WoT service mashups, for instance the *SAP Manufacturing Integration and Intelligence (MII)* ERP system [79]. However, many

of them suffer from drawbacks with respect to the formation of insular “walled garden” solutions and, similar to BPEL, regarding their very limited support for semantic service descriptions – for instance, EMMML merely provides an attribute that can be used to specify service functionality in a way similar to the hRESTS Microformat.

In the remainder of this section, we discuss several semi-automatic and a few automatic service composition tools to highlight the shortcomings of current approaches, especially in pervasive computing scenarios. We will conclude that current systems suffer from a lack of adaptability – especially given highly dynamic environments – and that it is not yet clear how REST service APIs should be described to enable the automatic composition of Web services. Given these observations, we propose two techniques to service composition: one of these is a semi-automatic, crowd-sourced approach (Section 7.2) and the other is a fully automatic mechanism that makes use of a lightweight form of functional semantic metadata to describe services in smart environments (Section 7.3).

7.1.3 Semi-automatic Service Composition Systems

As mentioned above, many of the service composition systems that are presented in the literature and widely used in industry feature a process-driven paradigm, meaning that they support end users (i.e., process designers) in composing service mashups using a domain-specific language or an abstract model of the composite service [7, 211], or by manually connecting or stacking service representations [191]. From the user input, these systems then create executable service specifications, for instance in BPEL [207]. Some approaches allow for runtime adaptability, meaning that abstract service placeholders in the process schema are replaced by service instances only at runtime, for instance based on non-functional properties such as QoS parameters [119]. For supporting the service composition process, some of these composition tools suggest appropriate individual services at the time of designing the mashup based on syntactic or semantic service properties [35]. Also, during the past few years, public “Mashup Tools” have been created, for instance Yahoo! Pipes [335], while others (e.g., Google Mashup Editor and IBM Mashup Center) have already been discontinued due to a lack of demand. A currently very popular service for creating simple two-stage service mashups is *If This Then That (IFTTT)* [288] that already contains more than 100 elementary services (e.g., Dropbox, Google Calendar, and Stocks and News services) and is very quick to take up novel services and devices that can be used in compositions: for instance, it provides connectors to *Google Glass* [215], *Belkin WeMo* home automation systems [250], and to *Nest* smart thermostats [307].

In the literature, several approaches to facilitate service composition on a larger scope and for business applications are proposed: [148, 207] contain current and very extensive discussions of more than 20 semi-automatic service composition approaches. While many of these systems provide very limited – often only syntactic – support during service selection, some – METEOR-S [210], WebDG [147], SOA4All [112], and the approach presented in [62] stand out: these provide semantics-based assistance to users when designing composite services, meaning that they categorize services using ontologies to help service designers quickly select appropriate services. METEOR-S adds semantic support to the METEOR workflow management system by making use of SAWSDL within the

annotations of individual services. The approach presented in [62] synthesizes composite services from a description of the desired functionality in natural language (e.g., “Print direction from home to restaurant”). The SOA4All project [112] represents a large-scale project within the European Commission’s *Seventh Framework Programme (FP7)* that aims to facilitate service discovery, mediation, and composition by adopting Semantic Web technologies in its core design: here, the design process consists of iterative refinement steps of a draft workflow that is created by the mashup designer – the designer and the automated semantics-based composition engine take turns to further refine the composition plan in each step of the process.

Many current mashup editors from academia and almost all prominent tools used for service composition in business environments feature visual composition interfaces to make them usable by mashup designers without programming or scripting skills [207]: for instance, in [94], objects and associated actions (e.g., the ringing of an alarm clock) are encapsulated in so-called *artifacts* that the user may combine in a graphical editor that also assists the user in debugging and supervising the resulting “Gadgetworlds.” Visual programming abstractions have also been applied in the context of facilitating the configuration of smart environments for end users in home automation scenarios [90, 191] – these systems enable users to create mashups that integrate functionality across services that are provided by multiple devices present in smart homes. To accomplish this, end users stack blocks that represent individual services in [191] or connect pictures of smart objects to describe the desired composite functionality [228]. Some of these studies included user evaluations and conclude that all of the participants familiarized themselves quickly (i.e., within a few minutes) with the concepts and were able to create applications for themselves with only negligible prior training. A commercial platform that enables users to create composite services in the home automation domain is Ninja [310] which, similarly to IFTTT, can incorporate many commercial platforms but enables users to create compositions that involve more than two services. Specifically for home automation in IoT scenarios that involve CoAP-based devices, the oBeliX tool [93] has been proposed, and also the ClickScript [305] visual programming language has been extended to handle smart things in the WoT [75, 138] and resource-constrained devices [120]. Visual abstractions have also been used in formerly academic tools that aimed to exploit the REST principles as a basic service composition mechanism, for instance in *JOpera* [173], a composition tool for SOAP and RESTful Web Services that can also execute the modeled workflows and is available as a plug-in for the Eclipse IDE. Likewise, Bite [198] proposes a similar BPEL-inspired composition language that builds on top of the REST uniform interface constraints. Finally, a highly intriguing approach to semi-automatic service composition that allows users to create simple device mashups using marker-based tracking and an augmented reality overlay on a handheld device is presented in [87].

However, according to [207], many of the proposed semi-automatic approaches in fact require mashup designers to *manually select* those elementary services that should be part of a composition, although some (e.g., [112, 147, 210]) provide syntactic or semantic selection support. The main shortcoming of these is, however, that they only provide very limited support for runtime adaptation of the composite services: the tools help users to

create static links between elementary services that cannot adapt in case services become unavailable or new services appear. An example to demonstrate this shortfall is JOpera, where the created visual models can automatically be compiled to Java bytecode: this is certainly beneficial with respect to the fast prototyping and deployment of mashups, but prevents runtime adaptation. On a different level, criticism of end-user programming in the form that is predominant in mashup tools today concerns insufficient security mechanisms in user-designed mashups [85], in particular when creating applications in IoT scenarios – this is true, and calls for the inclusion of non-functional properties such as QoS and security parameters in service definitions and, consequently, for considering these characteristics in the tools that compose them (this is also proposed, for instance, in the SOA4All project [112]).

7.1.4 Approaches to Fully Automatic Service Composition

To overcome the shortcomings of semi-automatic composition approaches with respect to runtime adaptability of the created services and to further facilitate the creation of mashups, the past decade has seen several systems that enable the fully automatic composition of elementary services ([148] and [207] contain excellent overviews of automatic composition tools). These typically rely on automated planning and scheduling techniques from artificial intelligence research or on different ways of semantically describing the “mini-world” of the individual services, for instance using techniques and languages from the Semantic Web or tailored solutions [207]: approaches such as OWLS-Xplan [104] and the approach presented in [144] are based on the Planning Domain Definition Language (PDDL) [145] that is used to define the local world model of the composite application and can be computed from the OWL-S descriptions of the individual services,⁴ while the SWORD framework [180] makes use of an Entity-Relationship world model. Together with a user-defined planning query, these systems then generate a composition plan for the individual services. Another interesting solution that is similar to what we propose in Section 7.3 for the composition of REST services in smart environments is [146] which uses the GOLOG logic programming language to construct composite services from primitive actions – an extension of this system that considers non-functional constraints such as user preferences is presented in [213]. To our knowledge, none of the proposed fully automated service composition solutions are in use in industry, although major research initiatives are targeting their deployment in this context [216].

The fully automatic composition of services still represents an open challenge which is, according to [207], largely due to a lack in interoperability between the different approaches, and the limited application domains of individual systems: many of the currently proposed systems that automatically create mashups from individual services do so by using planning languages such as PDDL that are limited regarding their expressibility across different domains. Those approaches that in principle make use of widely used Semantic Web technologies such as OWL-S often fall back to such languages for the planning step. Furthermore, many of the automatic composition systems suffer from the same

⁴An instructive example of how PDDL can be used in this context is shown in [325].

drawback as the semi-automatic approaches in that they cannot adapt to highly dynamic environments – context dynamicity, however, should be considered the default rather than an exception in pervasive computing scenarios. This is especially true when targeting applications on mobile devices whose entire smart environment changes when they are on the move. This disadvantage of current service composition systems is also mentioned by [207] as one of the main currently open issues regarding future research in the service composition domain: future composition systems should be adaptable to open and dynamic environments, ideally supporting self-configuration and self-adaptation as well as automatic optimization relative to the current environment and QoS constraints [33].

Finally, many of the currently available systems are based on description languages such as WSDL (and its semantic extensions) that have been conceived for the purpose of describing service-oriented architectures and only provide scant support for REST [207, 245]. In our opinion, this represents a missed opportunity since REST itself already comes with defined low-level protocol semantics and semantic descriptions for REST services could therefore focus on specifying the high-level functionality of a service and non-functional characteristics to create a more lightweight automatic composition system. To achieve this, however, it is not sufficient to annotate Web resources with “hints” about the functionality they provide (as proposed in the **hRESTS** Microformat) – rather, annotations should contain explicit machine-readable functional service descriptions [231]. Furthermore, an explicit state handling mechanism is crucial when considering applications that provide background assistance in smart environments, for instance with respect to configuring a user’s surroundings: the system must be able to automatically translate the current state of a resource (e.g., the local temperature) to semantic facts.⁵

In the rest of this chapter, we discuss two approaches of how we propose to address these challenges in highly dynamic Web-based pervasive computing environments. In the first, presented in Section 7.2, we aim to leverage the REST HATEOAS constraint for service composition wherever possible and created a composition service that we call a “computational marketplace.” This system depends on developers creating and publishing information about their mashups and thus represents a semi-automatic composition mechanism. Then, in Section 7.3, we propose an approach that enables the automatic goal-driven composition of Web services in ubiquitous computing settings. This system is based on a functional semantic service description language called RESTdesc [232] that combines a description of the REST API of a Web service with information about its high-level domain semantics. Finally, because the formulation of semantic goals is challenging for end users, we propose a method to facilitate the *creation of goals* using a graphical modeling language in Section 7.4.

⁵The authors of [207] also mention that explicit state management is required in future REST service composition systems. However, they refer to the *application state* while we do not believe that this aspect of REST environments must be explicitly handled. Rather, our state management targets the *state of resources* that are part of service mashups in pervasive computing scenarios (i.e., devices and their services), such as the current temperature of a specific room.

7.2 A Computational Marketplace for REST Services

In this section, we present a framework that links distributed computations within service mashups that are capable of distilling higher-level information from raw data, and in turn to further refine this information in a series of cascaded processing steps. We call our system a *computational marketplace* since it enables algorithms from different providers to interact when processing data in an open fashion and to compete with each other when multiple offerings provide the same kind of service. The focus of our marketplace is similar to services such as Yahoo! Pipes that was created to enable Really Simple Syndication (RSS) providers to aggregate and filter RSS feeds, but we focus on providers of Web services that perform computations rather than on information providers. It is furthermore different from marketplaces for applications (e.g., for smartphones or tablets) because, although its basic building blocks are individual service offerings, its purpose is the aggregation of distributed computations to yield service mashups that provide higher-level functionality. Our marketplace is an implementation of a semi-automatic service composition framework for REST services – other than traditional semi-automatic composition systems, it is, however, not process-driven. Rather, our approach is based on a crowd of developers who publish information about service mashups that they create. The purpose of our marketplace is then to enable machine clients to obtain this information and use it to use the published mashups.

When designing the computational marketplace, our goal was to adhere to the REST architectural style as closely as possible: we aimed at exploring to what extent the REST constraints – in particular, the HATEOAS interface constraint – are compatible with an open composition framework for Web services. In the subsequent sections, we present an analysis of the key features that any such framework should have (Section 7.2.1) and then discuss our concrete implementation (Section 7.2.2). Finally, we present the results from an evaluation of our system in a use case scenario (Section 7.2.4), and conclude with a discussion of the compatibility of the REST constraints and their relation to the automatic composition of REST Web services (Section 7.2.5).

7.2.1 Desired Features of a Computational Marketplace

We require a computational marketplace to be able to openly link distributed Web services while retaining scalability and remaining tolerant to faults of individual services and changes in how the individual computations are performed. Service directories such as *ProgrammableWeb* allow clients to find algorithms that suit their needs, but not to link them. Mashup engines such as *Yahoo! Pipes* and *JOpera* [173], on the other hand, allow to link REST services, but – due to their centralized nature – not in an open and scalable way. To ensure that our system achieves all these properties at the same time, we propose a set of architectural constraints on computational marketplaces.

7.2.1.1 Interface Discovery

First, a computational marketplace requires a way of advertising services to make them discoverable by clients. While centralized models such as, for instance, UDDI, rely on

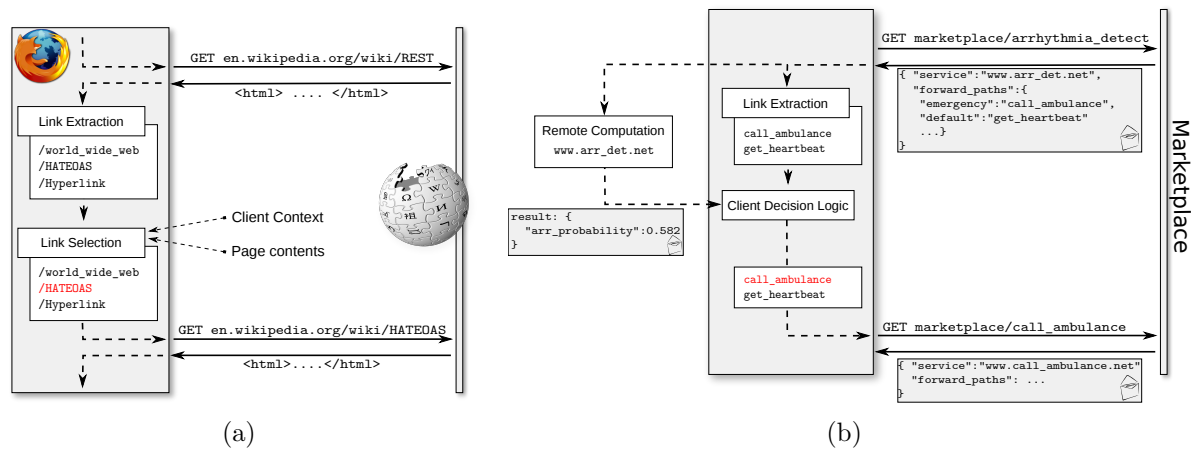


Figure 7.2: Comparison of clients that browse hypermedia structures: (a) A user who browses Wikipedia by following hypermedia links. (b) A client that executes a service mashup by following references that are provided by a computational marketplace.

services being registered, we have designed our system in a way that allows clients to discover services by following references: similar to websites that become discoverable as soon as they are linked to by other sites, service discovery in a computational marketplace is enabled by the fact that the individual services are hyperlinked as soon as they are involved in a composite computation.

7.2.1.2 Path Traversal Guidance

Second, we require a computational marketplace to be organized in a way that harbors and exposes all possible computational paths between individual service offerings, but leaves the decision of which concrete path a traversing client takes (i.e., which concrete services it uses within its mashup) up to that client. Thus, mimicking the HATEOAS constraint in the REST architectural style, we want to ensure that clients do not have to implement any knowledge about a mashup other than the ability to follow references that are provided by the marketplace. This is crucial to avoid tight coupling between clients and computational service offerings. In our system, the server guides the client through its traversal of the computational mashup, and the client only has to take local decisions regarding the selection of one of the provided links from one service to the next. As for traditional Web interactions, a change of a service interface is thus transparent to the client, who is merely waiting for hypermedia controls that guide it through possible state transitions and is thus tolerant to changes in the hypermedia structure.

Fig. 7.2 contrasts the way of how a human user traverses a traditional website to the interaction of a client with a computational marketplace. As discussed already in Chapter 2, users navigate websites by interpreting the information included in the representation of a requested resource to select the next hyperlink while taking into account their current context, including the page contents. Likewise, a client of a computational marketplace requests the representations of individual nodes in our system, where each node represents a Web service. The information that the marketplace returns about an

individual node contains the URI that can be used to invoke the underlying Web service as well as several *forward paths* (i.e., links to information about further computations) that it may follow after invoking the current service. In the example in Fig. 7.2(b), a client invokes an arrhythmia detection algorithm and uses the result that is returned by this algorithm in conjunction with its local decision-making process (i.e., its “context” – in the example, a threshold value) to select the path to follow: in this case, since the detector returns a rather high probability for a dysfunction, the client decides to invoke the service called `call_ambulance` next. Modifications to the paths that are exposed by the computational marketplace do not affect the ability of the client to use a composite application: for instance, our approach allows for the API of the `call_ambulance` service to change and for intermediate computations to be dynamically added or removed.

7.2.1.3 Self-Similarity & Statelessness

Third, a computational marketplace must exhibit horizontal and vertical flexibility. *Horizontal flexibility* refers to the ability of the system to create service mashups that exhibit the same structural properties as their constituents, meaning that these combined structures can be further combined with more modules (i.e., service mashups must be “self-similar”). *Vertical flexibility* refers to the ability of the marketplace to add and shed services that are equal from a computational point of view. This implies that, similar to what the REST architectural style requires for the communication between clients and servers in the traditional Web, the computations performed by the individual computations on a computational marketplace must be stateless: all information relevant for a specific service invocation must be passed along with the client request. If a service does require state, then it must be addressable by a URI which is passed along with the client request so that issues such as load shedding and state migration do not limit the scalability of the marketplace. This property is also key to the critical aspects of load balancing, recovery from failure, and dynamic resource allocation. It is also required for our system to be considered a *market* for computations, as clients should be able to switch freely between providers of the same kind of service.

7.2.1.4 Computational Paths Optimization, Security, and Billing

Additionally to the interface discovery, traversal guidance, and horizontal and vertical flexibility, we define a number of optional properties of a computational marketplace: these are not required for a marketplace implementation to fulfill its core purpose of exposing individual services and enabling clients to create and publish service mashups, but rather offer added functionality on top of these abilities that could be valuable to clients.

We believe that a computational marketplace should provide a mechanism that allows clients to *optimize computational paths* in accordance with the non-functional requirements of their application (e.g., with respect to the speed, money cost, or accuracy of services that are involved in a mashup). Following the HATEOAS principle, the marketplace should enable this by providing information related to these qualities of individual computations rather than deciding on behalf of its clients which paths are best. In this

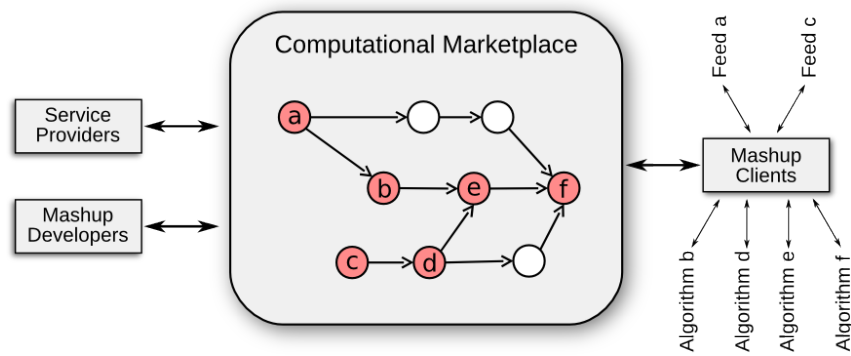


Figure 7.3: Abstract representation of a computational marketplace. The marketplace hosts services that are represented by individual vertices and connected by edges which represent links between services. It provides interfaces to clients, mashup developers, and service providers. The marked nodes constitute an individual mashup on the marketplace.

way, clients retain control about which paths are followed and can switch to other providers if the latency, cost, or reliability of a specific service changes.

Finally, since computational resources often represent an expensive utility and are secured by authentication and authorization schemes to restrict their use to an exclusive set of customers, a computational marketplace should accommodate security and billing mechanisms. Ideally, however, these tasks do not have to be handled by the marketplace itself – rather, service providers should incorporate a third-party billing and security scheme such as OAuth 2.0 [293] that can themselves be published as mashable Web services.

7.2.2 A Computational Marketplace for the Web of Things

In this section, we present a concrete implementation of a computational marketplace that was created in accordance with the above-mentioned constraints. An overview of our marketplace and its individual components is given in Fig. 7.3 – the marketplace is responsible for harboring the computational graph whose vertices represent individual Web services and whose edges represent paths between these computations – in the traditional Web, these vertices and edges would correspond to individual websites and the hyperlinks between them. If, in the marketplace, there is an edge between two vertices, then there exists at least one mashup that uses the output of the service represented by the source vertex of the edge as input for the service represented by its target vertex. Our marketplace itself is a Web service that stores information about all registered computational mashups and provides interfaces for clients who wish to traverse mashup graphs as well as for service providers and mashup developers that wish to expose information about individual and composite services, respectively. It thus provides functionality to three interacting parties: *service providers*, *mashup developers*, and *mashup clients*.

Service providers Providers of Web services may use the system to publish the APIs of their services along with API descriptions to support clients when using their services, where each service is modeled as a vertex resource.

Mashup developers Developers make use of the marketplace to define computational mashups on top of the individual services and to publish information about these mashups. The marketplace models such routes for computations by creating and exposing edge resources which imply that the involved computations are related – thus, each mashup constitutes a subgraph of the marketplace graph (e.g., the marked nodes in Fig. 7.3). Whenever a client executes a service within a service mashup whose corresponding vertex resource has an outgoing edge, the marketplace will signal that client that a related computation exists on its path. To specify how the output from one service should be mapped to the input of another, our implementation allows adding so-called *transformation maps* to edge resources. These maps introduce a certain degree of automation for simple and recurring input-output-mappings such as the renaming of data items or simple type conversions and, thus, convenience for mashup developers and clients. More complicated transformations should, however, be exposed on the marketplace in the form of Web services that are linked to the “worker” services.

Mashup clients Clients execute service mashups by letting the marketplace guide them through the graph structure along its edges. They choose which mashup to execute by selecting one of the subgraphs published by a mashup developer via the marketplace and then invoke the individual services of this mashup themselves, in a distributed manner. To enable this, the marketplace exposes the entry vertex for each registered mashup graph – this information can be used directly by clients to find entry points to service mashups and by external crawler components (i.e., “mashup search engines”) to discover individual mashups and services. Clients then use the marketplace to find a path from the most recently used service to the next within the context of their selected mashup, until they reach the terminal service of that mashup. For each service, our implementation furthermore provides them with information about non-functional characteristics of that node regarding several cost metrics (i.e., latency, accuracy rating, and money cost) that they can consider when selecting the next service. This gives them the opportunity to choose the fastest – or cheapest – computational path within their mashup and to dynamically adapt their traversal when paths become slower or more expensive. The implementation of a client of our marketplace is discussed in more detail in Section 7.2.3.

Security and Billing Our implementation does not explicitly take care of service access management and billing. However, these functions can be exposed as services on the marketplace and inlined as vertex resources within mashups, in the same manner as other Web services are used in the context of a mashup. To demonstrate this property, the use case shown in Section 7.2.4 includes a vertex resource that guides a traversing client to an OAuth-based authentication procedure before it can use a specific Web service.

7.2.3 Client Component

The client component is responsible for interacting with our marketplace on behalf of the client. In the traditional Web, the “client component” for human users is the browser that communicates with servers on behalf of users and renders representations of Web


```

public class ArrhythmiaDecider implements PathDecider {
    private static Gson gson = new Gson();

    @Override
    public String choosePath(String responseEntity, Map<String, ForwardPath> possibleForwardPaths) {
        // Unpack and analyze response entity
        ArrhythmiaResultBean arrhythmiaReply = gson.fromJson(responseEntity, ArrhythmiaResultBean.class);
        double probabilisticDecision = arrhythmiaReply.getProbabilisticDecision();

        String chosenForwardPath = possibleForwardPaths.get("start").getPathID(); // Restart by default

        if(probabilisticDecision > 0.5)
            chosenForwardPath = possibleForwardPaths.get("dispatch").getPathID(); // Dispatch if arrhythmia detected

        return chosenForwardPath;
    }
}

```

Figure 7.4: A tailored traversal manager for a specific mashup: the client selects the path called “dispatch” whenever the reply from an invoked algorithm indicates an arrhythmia probability of more than 50%. Else, it selects the path called “start.”

```

public class LowestTimeCostDecider implements PathDecider {

    @Override
    public String choosePath(String responseEntity, Map<String, ForwardPath> possibleForwardPaths) {
        ForwardPath firstPath = possibleForwardPaths.values().iterator().next();
        String minimumTimePath = firstPath.getPathID();
        double minimumTime = firstPath.getCostMetric().getTimeCost();

        // Select path with lowest time cost
        for (ForwardPath forwardPath : possibleForwardPaths.values()) {
            double timeCost = forwardPath.getCostMetric().getTimeCost();
            String pathID = forwardPath.getPathID();

            if(timeCost < minimumTime) {
                minimumTime = timeCost;
                minimumTimePath = pathID;
            }
        }

        return minimumTimePath;
    }
}

```

Figure 7.5: A generic traversal manager that always selects the path with the lowest latency.

resources and, specifically, the links contained therein, graphically. Similarly, a client component of a computational marketplace handles responses from the marketplace that indicate vertex resources which can be reached from the client’s current node within the mashup that it is traversing. Consequently, the main function of the client component is to *select* one of the vertices that are exposed by the marketplace and thereby control the traversal on behalf of the client.

In its simplest form, the client component could, for each step that it takes in a mashup, ask the user for feedback about which link to follow next – the resulting software would appear to humans like a “browser” for distributed Web services that would, after each step, display the result of the remote computation and a choice of paths that it could follow. Since, however, our goal is to have the selection of an appropriate next service be done in an automatic way, our implementation of a client component contains the concept of *traversal managers* that are programmed by the user to traverse the graph in the desired manner. Note that the marketplace itself cannot know details about the user’s decision-making process – therefore, the concrete implementations of the traversal managers are fully decoupled from the marketplace and have to be written by the individual wishing

to use the marketplace, where usually also the results from preceding computations are taken into account when deciding which path to follow. For instance, Fig. 7.4 shows a tailored traversal manager that selects one of the paths “start” or “dispatch” depending on the response from a service execution. In our prototypes, however, we often make use of simple generic traversal managers that, for instance, always select the path that has the lowest latency as announced by the marketplace (see Fig. 7.5).

7.2.4 Evaluation

We tested our implementation of a computational marketplace on a use case where a complex real-world coordination problem is solved by a mashup of distributed Web services: Our service mashup accesses real-time feeds of heartbeat data and analyzes these to detect signs of arrhythmia. In case a dysfunction is detected, the client that traverses the mashup invokes another service which dispatches an ambulance to the patient’s current location. To optimize the dispatch process, this service takes into account information about driving times from multiple hospital locations to the patient. Driving times are modeled as dependent on the amount of traffic which in turn depends on the current weather conditions – for this reason, information from a weather feed is taken into account when estimating the current traffic for dispatching the closest ambulance to take the patient to the most appropriate hospital. Of all services that are used within this mashup, only the arrhythmia detector and the ambulance path optimization service were developed by ourselves. For all other functions as well as for visualization and persistence, publicly available Web services such as the Google Prediction API [277] and the Yahoo! Weather API [336] were used, which demonstrates the compatibility of our system with current Web service APIs that are in widespread use. To illustrate the provisioning of authentication and authorization services, a client that uses the Google Predict API in the process of traversing the service mashup is directed to an intermediary node that uses OAuth to authorize the client’s access using its Google credentials.

7.2.4.1 The Arrhythmia/Ambulance Dispatch Mashup

The complete service mashup for this use case consists of four mashups that implement its individual components. In the composite scenario, all these mashups, that are represented as graphs on the computational marketplace, are combined to incorporate refined weather and traffic information with the arrhythmia detection services to optimize the ambulance dispatch process:

- The *Arrhythmia Mashup* decides whether a patient is suffering an arrhythmia. Its main components are a feed that fetches heartbeat data from patients and two arrhythmia detection services. To simulate heartbeat data, we use samples from the database on malignant heart rhythms from the Boston Beth Israel Deaconess Medical Center [67].
- The *Weather Mashup* uses the Google Predict API to output one of 47 high-level descriptions of the weather conditions (e.g., “partly cloudy”), where the raw weather

data (temperature, humidity, etc.) is obtained from the Yahoo! Weather API. Accesses to the Google Predict API are authenticated and authorized via OAuth.

- The *Traffic Mashup* combines simulated geo-location information of cars and pedestrians with the weather prediction to create traffic density estimates.
- The *Dispatch Mashup* combines location information from simulated patients, ambulances, and hospitals with traffic density estimates to dispatch ambulances to patients. To obtain driving information, it uses the Google Maps API.

To expose information about the entire mashup on our computational marketplace, a *service provider* first uses the marketplace to create vertices that represent the individual services. For each service (e.g., an arrhythmia detector), the provider at least specifies the URI of the corresponding Web service, the REST method to use the service, and the output media type of the service. Additionally, if required, request headers and parameters for specifying query, form, and path inputs may be given. Next, a *mashup developer* links the individual vertices and specifies the input/output-mappings between the algorithms, either by using the generic transformation maps provided by the marketplace or by creating computational vertices that handle the transformations. Finally, mashup clients can navigate the exposed mashups to implement the complete *Arrhythmia/Ambulance Dispatch* mashup.

7.2.4.2 Dynamic Optimization

We used one of the constituents of this service mashup – the Arrhythmia Mashup – to demonstrate the dynamic optimization capabilities of our computational marketplace: by exposing information about non-functional properties of the individual services, the marketplace enables clients to dynamically switch between providers according to their requirements with respect to the money cost, latency, and accuracy of a service. To illustrate this, we added three dummy arrhythmia detectors to the marketplace and configured them to exhibit time and money costs that grow linearly with the number of clients each detector is servicing at a certain point in time.⁶ Furthermore, we configured several clients with different non-functional requirements and thus expect that these should dynamically adapt to the changing costs by routing their executions via services that best suit their requirements.

The results of a test case where 15 different clients concurrently use the arrhythmia detectors demonstrate that the dynamic distributed optimization also works in practice: of these clients, three always choose the same detector (i.e., each chooses one of the available three), seven always request the *cheapest* computational path, and five request the *fastest* service. Fig. 7.6, which shows the number of clients being serviced by each of the three detectors over time, shows that the clients indeed route their computations along different paths whenever a peak load is reached on one of the detectors as, for instance, for *Detector 1* at iteration 350.

⁶As an example, suppose that *Detector 2* has a base latency (or money cost) of x ms for a single computation and is currently servicing two clients. A third client that now makes a request to *Detector 2* will have to wait for $3 \cdot x$ ms (or pay $3 \cdot x$ money units, respectively) before the request is answered.

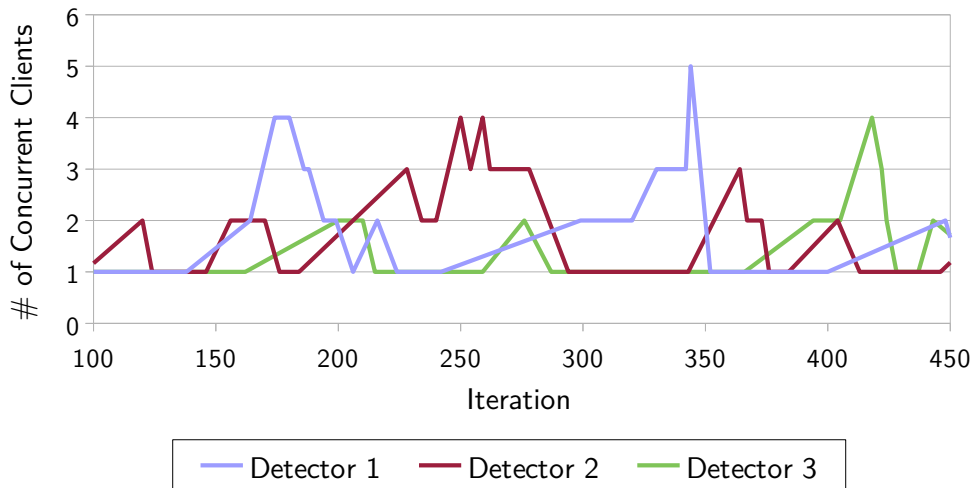


Figure 7.6: Illustration of the dynamic optimization capabilities of our marketplace implementation: 15 clients interact with three arrhythmia detection services. Of these 15 clients, three always select the same detector, seven request the cheapest detector, and five request the service with the lowest latency.

This evaluation shows that our computational marketplace is indeed able to balance client demands and provider offerings in a form of dynamic equilibrium that is reached through decentralized decision making. It does, however, not attain an optimal solution as we opted for a decoupling of clients, services, and the marketplace to be compliant with the REST HATEOAS principle of not controlling the traversal of clients but rather guiding them by providing options for their next action in the form of hyperlinks.⁷

7.2.5 Service Composition and HATEOAS

Our computational marketplace implementation represents a semi-automatic service composition platform that builds heavily on the REST HATEOAS constraint for linking computations from distributed providers within service mashups, and for guiding clients when traversing the resulting service graphs. The proposed platform, however, has several disadvantages that make it unsuitable as a basis for creating a fully automatic service composition framework – in this section, we discuss these shortcomings and attempt to link them to core characteristics of the HATEOAS principle. We will conclude that HATEOAS, by itself, is unsuitable to achieve automatic service composition and that additional information about service mashups is required on top of it for guiding machine clients within hypermedia mashups.

Both JOpera [173] and Bite [198] go beyond harboring the links between computations and also invoke computations themselves – this inhibits their scalability as all service invocations are performed by a central instance on behalf of clients, and represents a violation of the HATEOAS constraint: clients are not merely guided while executing a service mashup, but rather the service execution is performed *by the composition framework*, on their behalf. To retain core REST principles, others have proposed to separate the con-

⁷Assigning all clients statically to a specific detector would reduce the overall waiting time.

cerns of service composition and the description of links between services: The authors of [5] use Petri Nets as a basic composition model and augment them with descriptions based on the Resource Linking Language (ReLL) [4] to expose meta-information about the linkage of the individual services for crawling and semantic integration. Also the authors of [17] emphasize linking as an integral part of composing resources on the Web and guiding RESTful machine-to-machine interaction. Also our concept of a computational marketplace also satisfies the HATEOAS constraint by providing links to related services within the composition and not requiring a central instance to guide clients. Our system, however, suffers from several shortcomings that become obvious when examining how exactly clients choose a service mashup to execute, and in how they make the local decisions that drive their traversing of the service graph:

To *select* a mashup, clients use that mashup's URI. This means that we encode the entire functionality that a mashup provides to a client within its URI – to achieve complete automation, a machine client would need a way of parsing the description of an entire mashup to find out whether it implements the desired functionality. While this is possible in principle, it is reminiscent of how clients use WSDL descriptions to infer how to invoke a service. However, this information must be delivered in one way or another because a method of describing the purpose of an entire service mashup is necessary to enable clients to select appropriate mashups to reach their application goal: without this global information, which is neither conveyed within hypermedia nor by the semantic link annotations provided by vertex resources in our implementation nor in the approaches mentioned above ([5] and [17]), a client is not able to make local decisions regarding which service to execute next. To enable this, the client requires “global” guidance on top of the “local” guidance that HATEOAS and annotated hyperlinks provide. The reason for that requirement is that machines cannot, in fact, “follow their nose” within hypermedia systems, even if individual links are semantically annotated. To give a simple example of this, imagine a machine client that intends to “buy the book with ISIN X ” (this goal is, for instance, programmed by a human). Upon reaching the Web resource that represents that book, the client will perhaps – in accordance with HATEOAS – be able to discover a semantically annotated link that allows it to “put the book with ISIN X into the shopping cart.” However, the client is, under these circumstances, unable to link this “immediate goal” to its “global goal” of buying the book – to do this, it requires global information that the partial goal will help it to achieve its global goal.⁸

One way of enabling true goal-driven behavior for machine clients in this scenario would be to annotate *each intermediate link* within a composite service with all goals that might be reached by following this link – this is *exactly* what our computational marketplace does: each registered mashup graph holds information about all its constituent nodes and edges. When asking our marketplace for forward paths from a node that represents a specific service, mashup clients include information about the mashup that they are currently traversing (e.g., the URI of the Arrhythmia Mashup) and the marketplace returns only edges that are relevant in the context of this global goal of the

⁸The same is true for humans. For them, however, “put[ing] the book with ISIN X into the shopping cart” already conveys the affordance that the book can subsequently be bought because the system is modeled on how people shop in the real world.

client. In other words, in our marketplace and other similar systems, REST and HATEOAS merely provide information about how to *traverse* the underlying hyperspace, but not about the global high-level semantics of following a hyperlink. In the traditional “human” Web, browsers know how to traverse the hyperspace thanks to the generally agreed REST architectural style, but humans interpret the contents of websites and hyperlinks contained therein. Similarly, REST by itself does not allow machine clients to automatically traverse hyperspace and reach a final goal by themselves – rather, they require a supporting system that enables them to make the correct local decisions to reach a global goal.

It is also interesting to note at this point that our computational marketplace represents a layer of hypermedia links on top of that defined within the involved resources that represent individual services. This is necessary for coordinating mashup executions across the boundaries of service providers because otherwise, every single service resource would have to embed links to all potentially related services within its representation which is, in our opinion, infeasible. This practice of linking services via the marketplace could be interpreted as a violation of the HATEOAS principle (since the links are not under the control of the server that hosts the service), but we believe that it is our only option to enable collaboration between globally distributed services in this context.

Another shortfall of our computational marketplace is the way how clients reach local decisions: in our system, the marketplace provides them with a set of links to vertex resources that represent services which might be relevant given the current situation of the client. The client then uses local traversal managers to select the most appropriate link and continue traversing the mashup in that direction. However, we use plain keyword annotations to identify the different links which means that clients’ traversal managers are implemented and compiled against this rather arbitrary vocabulary (link annotations are created by mashup developers when defining a mashup graph). Consequently, while our implementation is tolerant with respect to changes of the underlying Web services, it cannot handle a change in the *name of an edge resource*. This represents an issue at the core of the Linked Data movement, where standardized vocabularies have been proposed to provide local link annotations.

In the following, we discuss a different approach to enable fully automatic service composition that uses technologies from the Semantic Web to describe the functionality provided by individual services. These descriptions are directly linked to the services instead of publishing them via a central marketplace and a semantic reasoner is used to infer a service mashup that achieves the client goal at runtime – because the reasoner can link immediate partial goals to the global application goal of a client, this system achieves true goal-driven behavior.

7.3 Semantics-based Service Composition

Our discussion of computational marketplaces and their shortcomings with respect to the global guidance of clients that traverse service mashups have shown that we can rely on REST, HATEOAS, and link annotations for enabling local decision-making as well

as a solid basis for specifying the protocol semantics of interactions with Web services. Our marketplace furthermore includes information about the global goal that a client can reach by executing a service mashup in the form of the mashup resource URIs – however, machine clients cannot process these annotations in a way that would allow them to select a suitable mashup given a user goal, and they are thus not suitable for achieving automatic service composition in WoT scenarios. In that sense, our computational marketplace can merely be used to link services on a syntactic level: it can propose service nodes that are suitable for processing certain input formats but cannot help a mashup developer in deciding whether it is *sensible* to apply a service to the input data at hand – this functionality is very similar to search engines such as Hoogle [306] that enable developers to search for implementations by specifying method signatures.

In this section, we discuss an approach to fully automatic service composition that allows clients to automatically create and traverse service compositions in WoT scenarios with the help of functional semantic annotations. These allow to reason about the capabilities of Web services and thereby enable the goal-driven configuration of smart environments for end users. To specify these high-level domain semantics of a service, we use the RESTdesc language [232] that we have extended to make it suitable for reasoning about service capabilities in smart environments. The advantage of this system for users of smart environments is that, instead of having to design a service mashup that achieves their goal or manually select a mashup in the computational marketplace, they merely have to state that goal in a machine-understandable way. Given this statement of a user's goal, a reasoning component in our system determines whether the goal can be reached given the set of available services, and also infers which user actions (i.e., HTTP requests in this case) are necessary to reach it. The user can then execute these requests and thereby modify his environment to reach the desired goal. Because service mashups are created at runtime from user goals, this approach exhibits a high degree of flexibility as service mashups can adapt to dynamic environments and are fault-tolerant with respect to individual services becoming unavailable, which – as we discussed in Section 7.1.2 – is particularly desirable in IoT and mobile scenarios [207]. However, since we cannot assume that users are familiar with semantic languages, it is necessary to provide an abstraction layer on top of the semantic descriptions through an interface that simplifies the goal creation for end users. We therefore integrated our system with a visual programming language that supports end users by enabling them to model the desired state of their smart environment graphically and thus hides the technicalities of the underlying semantics and the reasoning (see Section 7.4).

7.3.1 System Context

To enable the composition of services that are provided by devices in smart environments, our system must have access to a means of discovering these individual services and their associated semantic descriptions. Our service composition system thus requires a component that knows about services that are accessible for a client that wishes to create composite applications on top of them. For this purpose, we make use of the Web-based discovery and look-up infrastructure that we presented in Chapter 6 – our

composition system is, however, compatible with any search engine or service repository that enables clients to find the URIs of service endpoints: specifically, the software that we present in this section is compatible with discovery services that have been proposed by other research groups (e.g., [40]), and with search engines for the WoT such as Dyser [167]. When considering resource-constrained devices that support the CoAP protocol, our system can make use of the CoRE Resource Directory [294] as a service discovery component. Finally, industrial standards for home entertainment systems such as UPnP [289] are in principle also compatible with our system, given that it can gain access to the URIs of individual Web services in such settings.

Apart from a component to find services in a smart environment, our system requires access to a *semantic reasoner* to be able to infer the global structure of a service mashup from the semantic metadata that is provided by these individual services. In principle, this reasoner could be hosted on a remote server – however, we decided to use a local implementation in our prototype because of the higher delays when using a remote instance and privacy and security considerations. In the context of our proposed composition framework for services in smart environments, we consider the reasoner a trusted entity: it has access to functional metadata provided by all services in the environment, is aware of the user’s goals with respect to using these services, and eventually proposes HTTP requests to be executed by the user to achieve its goals.

Finally, our system requires a component that interacts with the reasoner and with services in the smart environment, on behalf of the user. This interface, that can be a Web application or a mobile application that is deployed on the user’s smartphone or tablet, is used by humans to formulate goals with respect to their environment. Given a goal, it queries the reasoner for a service mashup that allows to reach that goal and executes the requests proposed by the reasoner.

In the following, we discuss how these components collaborate to enable users to configure their smart environment. Our approach thus permits users to interact not only directly with individual devices in their surroundings but with composite applications that provide higher-level functions and can also include remote service offerings on the Web. One cornerstone of our system is the embedding of functional semantic metadata in the form of RESTdesc descriptions within the individual service endpoints. We extended RESTdesc with capabilities that make the language applicable to composition tasks in smart environments while preserving the logical integrity of the reasoning – we present this language and our proposed extension in Sections 7.3.2 and 7.3.3. In Section 7.3.4, we discuss the resulting service composition system for smart environments with respect to its scalability, expressibility, correctness, and usability for end users, and also review challenges that arise from the adoption of semantic technologies as its basis. Finally, in Section 7.4, we present how we integrated our approach with *ClickScript*, a visual programming tool that enables end users to formulate semantic goals which describe the desired state of their smart environment.

7.3.2 Semantic Metadata for REST Services

Because all smart devices and services that we consider feature Web APIs that are modeled according to the REST principles, their low-level protocol semantics are already specified by HTTP. On top of this, we define their high-level domain semantics (i.e., which function a service provides) using RESTdesc, a machine-interpretable functional service description format for REST APIs. RESTdesc descriptions are expressed in Notation3 (N3) [251], an RDF superset that adds support for quantification. Services expose these descriptions for automated discovery – thus advertising their functionality – by linking to RESTdesc documents using the `Link` HTTP header's `describedby` relation [292] as part of the responses to HTTP `GET` and `OPTIONS` requests. This mechanism enables any client that knows the URL of a service to download its semantic description. We illustrate the main concepts of RESTdesc using the example of a service that can convert temperatures given in degrees Celsius to Fahrenheit values whose semantic description is shown in Listing 7.1. Later, as a simple example of service composition in smart environments, we will show how a semantic reasoner can be used to automatically create a service mashup that combines the functionality of this converter with a smart thermostat that takes Fahrenheit values as input to configure its setpoint (Section 7.3.3).

At the highest level, a RESTdesc description consists of three parts: preconditions, postconditions, and an HTTP request that realizes the postconditions from the preconditions. In the example in Listing 7.1, the preconditions (lines 7 to 9) stipulate that a certain temperature expressed in degrees Celsius exists, and that this temperature has a specific value. The postconditions (lines 17 to 20) warrant that there exists a temperature expressed in degrees Fahrenheit that is the same as the Celsius temperature.

```

1  @prefix owl: <http://www.w3.org/2002/07/owl#>.
2  @prefix dbpedia: <http://dbpedia.org/resource/>.
3  @prefix ex: <http://example.org/#>.
4  @prefix http: <http://www.w3.org/2011/http#>.
5
6  {
7    ?tempC a dbpedia:Temperature;
8           ex:hasValue ?cVal;
9           ex:hasUnit "Celsius".
10 }
11 =>
12 {
13   _:request http:methodName "GET";
14             http:requestURI (<http://converter.net?temp=>?cVal>);
15             http:resp [ http:body ?fVal ].
16
17   _:tempF a dbpedia:Temperature;
18           ex:hasValue ?fVal;
19           ex:hasUnit "Fahrenheit";
20           owl:sameAs ?tempC.
21 }.

```

Listing 7.1: A RESTdesc description of a temperature conversion service.

```

1 @prefix dbpedia: <http://dbpedia.org/resource/>.
2 @prefix ex: <http://example.org/#>.
3
4 _:roomTemp a dbpedia:Temperature;
5             ex:hasValue "20";
6             ex:hasUnit "Celsius".
7
8 _:convertedTemp ex:hasValue ?value;
9                 ex:hasUnit "Fahrenheit".

```

Listing 7.2: A semantic goal that asks for a Fahrenheit temperature that is equivalent to the specified Celsius temperature.

Finally, the HTTP request (lines 13 to 15) is a **GET** request to a URL determined by the value of the `cVal` variable that returns the Fahrenheit value in the response body. This HTTP request is described by the *HTTP in RDF* vocabulary [328] which provides a semantic way to describe HTTP exchanges. The description as a whole communicates in a machine-interpretable way how a Celsius temperature can be converted to the equivalent Fahrenheit temperature.

In more detail, the basic unit in N3 is the *triple* that is expressed in the format “**Subject Predicate Object.**” N3 also has formulas which group together triples (between braces {}), variables that start with a question mark ?, and implications (i.e., triples where the predicate is `=>`). When multiple predicate-object pairs are separated using semicolons, all of these pairs are associated to the leading subject. For instance, lines 7 to 9 state that the variable `tempC` is a “Temperature,” that its relation to another variable, `cVal`, is “hasValue,” and that its relation to the constant `Celsius` is “hasUnit.” The `@prefix` declarations indicate which ontologies are used within a document and allow to abbreviate URLs of subjects, predicates, and objects. For instance, we use well-known public ontologies such as DBpedia [281] which has been created to make structured data in Wikipedia accessible for machine clients.

Because RESTdesc descriptions are regular N3 implications, they can be applied as inference rules by N3 reasoners without requiring any special support. For each rule it holds that, if the triples in the antecedent can be matched, the triples in the consequent can be concluded. To find out whether a specific goal can be reached in a given context, users can thus use a semantic reasoner that has access to service descriptions such as that of the temperature converter shown above. For instance, a user could ask which Fahrenheit temperature is equivalent to 20 degrees Celsius (Listing 7.2). Given this goal, a reasoner can instantiate the description of the temperature conversion service which will indicate that the answer is given by an HTTP **GET** request to the URL `http://converter.net?temp=20`. Thus, the reasoner does not immediately tell the result of the conversion but rather indicates that the client must execute the described HTTP request to convert the temperature value. When a reasoner has access to multiple rules, it can *chain the implications* they contain and thereby find out how the client must coordinate invocations of different services that can together achieve the user goal. For instance, if the user wants to set a temperature of 20 degrees Celsius in an environment that contains a smart thermostat which can only take inputs in degrees Fahrenheit,

the reasoner will instruct it to first send an HTTP `GET` to the converter service, unpack the response body, and send the obtained temperature value (in degrees Fahrenheit) to the thermostat. The combination of RESTdesc descriptions with reasoning thus yields a powerful service composition mechanism [230].

To summarize, the entire workflow in our prototype system is as follows: The reasoning service can either run locally on the client (if the client is sufficiently powerful) or be provided by the user interface device (e.g., a smartphone) or a remote server for resource-constrained clients. To obtain information about the individual services that are accessible to the client, the reasoner first does a look-up with our management infrastructure to find their URLs. It accesses these URLs using HTTP `OPTIONS` requests, follows the links that are returned within the `Link` header fields, and parses the `.n3` documents at these locations, thereby creating a local service catalog. When the user asks for a specific goal, the reasoner is invoked with the service descriptions from that catalog and the user's goal. Using backwards chaining, the reasoner then searches for a path from the current state to the goal state. If successful, it returns the necessary HTTP requests, which are executed by the client on behalf of the user to reach the goal.

7.3.3 Reasoning in Smart Environments

In the previous section, we described how RESTdesc can be used to semantically annotate REST service APIs and how a reasoner can infer whether – and how – a user goal can be reached by integrating functionality across services. However, it is not straightforward to apply RESTdesc in the context of the configuration of smart environments: the main issue when trying to integrate its semantic descriptions with our systems and implementing use cases from the field of pervasive computing is that RESTdesc – being grounded in first-order logic – is not able to distinguish between mutually exclusive *states* of components of the system (e.g., of a specific device in the user's smart environment). Therefore, while RESTdesc works very well for describing services that do not induce incompatible states such as the temperature converter in the previous section, already the most basic use cases that involve *stateful* objects cause problems regarding the soundness of the reasoning. As a simple example, assume the system has access to the fact that a room has a temperature of 23°C. If the user then defines a goal where the same room has a temperature of 22°C, this introduces a logical contradiction because no room can have two different temperatures at any given moment (note that it is not possible to *remove* facts from the knowledge of a first-order logic system).

For this reason, we extended RESTdesc by incorporating a mechanism that allows to explicitly describe states of smart environments, and of devices within smart environments. Furthermore, we introduced the concept of *state transitions* to enable the annotation of services that induce state changes, the semantics of which are described in a states ontology. This ontology is publicly available⁹ and can be looked up by reasoners for successful state handling.

As an example of a service that makes use of our states ontology, consider the REST-

⁹See <http://purl.org/restdesc/states>

```

1  @prefix st: <http://purl.org/restdesc/states#>.
2  @prefix log: <http://www.w3.org/2000/10/swap/log#>.
3  @prefix dbpedia: <http://dbpedia.org/resource/>.
4  @prefix ex: <http://example.org/#>.
5  @prefix http: <http://www.w3.org/2011/http#>.
6  @prefix geonames: <http://www.geonames.org/ontology#>.
7
8  {
9    ?newTemp a ex:Temperature;
10             ex:hasValue ?fVal;
11             ex:hasUnit "Fahrenheit".
12
13    ?thermostat a dbpedia:Thermostat;
14                geonames:locatedIn ?place.
15
16    ?state a st:State;
17            log:includes { ?place ex:hasTemp ?oldTemp. }.
18 }
19 =>
20 {
21   _:request http:methodName "PUT";
22             http:requestURI (?thermostat "?t=" ?fVal).
23
24   [ a st:StateChange; st:removed { ?place ex:hasTemp ?oldTemp. };
25     st:added { ?place ex:hasTemp ?newTemp. };
26     st:parent ?state ].
27 }.
```

Listing 7.3: A RESTdesc description of a temperature conversion service.

desc description of a smart thermostat in Listing 7.3. From the antecedent of this rule, we can see that an execution of the service requires a temperature value in degrees Fahrenheit (lines 9 to 11) as well as the presence of a device of type `Thermostat` at a specific location (lines 13 and 14). The preconditions furthermore contain a state description which specifies that the ambient temperature at the location of the thermostat is `?oldTemp` (lines 16 and 17). The consequent of the rule specifies that an HTTP PUT request to the thermostat (lines 21 and 22) will result in a state transition (lines 24 to 26): the new state does not anymore include the `?place` having a temperature of `?oldTemp`, but rather includes the new fact that the temperature at the location of the thermostat is `?newTemp`.

To find out how to set the ambient temperature at a specific location called “Office” to 23°C, a user would now formulate the goal shown in Listing 7.4. In this goal, the user first defines the `temp23` constant that includes the desired temperature value as well as the information that this value is given in degrees Celsius. This entity is then used when defining the desired state of the location “Office.” As described in Section 7.3.2, this goal can now be sent to a reasoner that will indicate that the goal state can be reached by first sending an HTTP GET request to the converter service that includes the Celsius value to obtain the corresponding Fahrenheit value, and then sending an HTTP PUT request to the URL of the thermostat at the location “Office.” Note that, because we the concrete

```
1 @prefix st: <http://purl.org/restdesc/states#>.
2 @prefix ex: <http://example.org/#>.
3
4 :temp23 a ex:Temperature;
5         ex:hasValue "23";
6         ex:hasUnit "Celsius".
7
8 ?state a st:State;
9         log:includes { :Office ex:hasTemp :temp23. }.
```

Listing 7.4: A RESTdesc description of a temperature conversion service.

location is dynamic in the service description in Listing 7.3, the URL of a correct smart thermostat is given by the `?thermostat` variable and found at runtime from all available thermostats in the system. Because our management infrastructure also features semantic descriptions of its functionality, our system is thus able to find out by itself that it can use the infrastructure to *locate an adequate smart thermostat*.

To summarize, we have successfully extended RESTdesc with the concepts of states and state changes. This enables using our system to describe any service that induces state transitions, and specifically to model states of smart environments. Based on our approach, it is now possible to create an application that runs on the user's smartphone and lets the user specify the desired state of different named locations (e.g., the user's office) with respect to properties such as the ambient temperature or the desired media playback – the reasoning system can use the same naming scheme as our management infrastructure to identify locations.

7.3.4 Discussion

We implemented several applications that demonstrate the capabilities of our semantic composition system for supporting end users in configuring their smart environment in the home automation domain: for instance, the *Music Escort* application is aware of the user's current location and music preferences and streams his favorite songs directly to media devices in the user's vicinity (see Fig. 7.7). The mashup involves many different devices and complex interactions between them and we use it to exemplify how our system deals with dynamic situations where information, such as the user's location, can become invalid and the client has to compensate [131]. Another prototype application, the *Room Configurator*, enable users to control not only the currently playing music at their location, but allows them to enter several comfort parameters such as their comfort temperature or preferred lighting level. The client interface, which has been implemented as a smartphone application, then communicates with the user's surroundings to implement these settings. The important commonality of both these systems is that they make no assumptions about which kinds of devices are available in the user's smart environment but merely assume that devices advertise their services using descriptions in the RESTdesc format that, in turn, can be processed by a semantic reasoner. In this way, both applications are in principle capable of interacting with many different types of smart things that provide the services they seek, on behalf of the user.

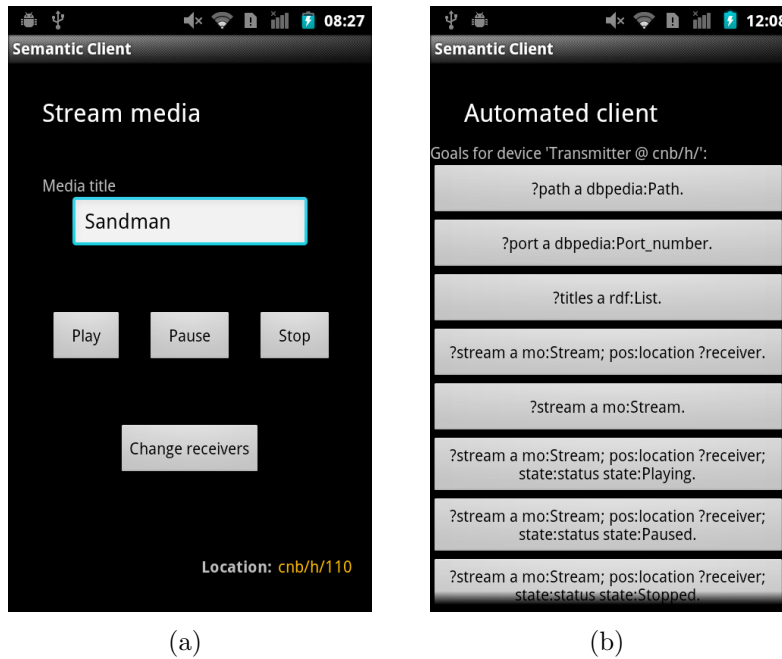


Figure 7.7: The *Music Escort* application: (a) The preferred song can be entered on a simple user interface. From this information, the application creates a semantic goal that includes the specified song playing at the current location of the user. This goal is sent to a semantic reasoner and the requests returned by the reasoner are executed by the application to realize the goal. (b) The application can furthermore display all currently reachable goals given the current smart environment of the user.

To further evaluate our proposed approach to automatic service composition in smart environments, we draw on earlier studies that assess service composition systems with respect to a set of qualities these systems should exhibit. [207] considers several requirements on composition approaches that are assigned to different phases in the life-cycle of a service mashup: In the mashup definition phase, the *expressibility* of the process modeling language and the *correctness* of the designed composite service are considered relevant. In the service selection phase (i.e., when the system attempts to locate suitable individual services and must arbitrate between similar service offerings), the degree of *automation* and the *selectability* according to non-functional properties matter. During the execution of the mashup, the relevant characteristics of a composition system are its run-time *adaptability*, *scalability*, whether it enables users to *monitor* the system, and the *reliability* of the execution, i.e., its robustness to exceptional behavior. Finally, [207] defines the overall requirements of *personalization* (i.e., the capability of the system to consider the user's context) and *tool support*, which refers to the availability of appropriate tools to support the creation of the service mashup and the managing and monitoring of the execution. Similarly, [243] considers properties such as the domain independence, correctness, semantic capability, QoS awareness, adaptability, and scalability of a semantic composition system – these parameters correspond to the characteristics set forth by [207] (e.g., domain independence and QoS awareness correspond to the expressibility and selectability requirements, respectively) but are specified on a coarser level.

By its very nature, our system achieves a *high degree of automation* with respect to the selection of individual services: given that these services are annotated appropriately, service selection and composition is done fully automatically. It is also *highly adaptable* with respect to the dynamic availability of specific services in a smart environment: the ability to bind services dynamically at runtime lies at the core of our system, and clients may additionally choose to re-query the reasoner in the middle of executing a service mashup (in extreme cases, after each service execution step) for maximal adaptation – this is supported especially since our approach allows to obtain execution plans very rapidly, compared to other concepts. Our system also features a *high level of personalization*: user preferences and context characteristics (e.g., the locations of smart thermostats) that are available to the reasoner as logical facts are automatically considered during the service composition phase. Since, in our system, a composite service mashup is not executed by a central execution engine, our system can also be considered to be *simple to monitor* by the client: the client executes all requests to individual services itself! This is closely tied to our system’s *reliability*: our approach does not automatically handle exceptional behavior, but the client is explicitly informed about incidents via HTTP status codes that are returned by the individual services – the recovery itself must, however, be implemented by the client itself. However, if the reason for a failure was a transient fault in the system, for instance related to bad connectivity, it might be sufficient to execute the HTTP requests once more – this is supported by the stateless nature of the protocol. Alternatively, if the reason for the fault was a component of the system that became unavailable, the reasoner should be asked again for a new service execution plan. Both these resolution strategies are generic and do not depend on an explicit fault handling mechanism. Finally, our system only provides *limited tool support*: we have integrated it with a visual programming language to facilitate the creation of user goals (see Section 7.4) and users can monitor service executions in real time using the tools that we presented in Chapter 5. In the following sections, we discuss our semantic service composition approach with respect to the remaining desirable qualities of such systems set forth by [207]: *scalability*, *expressibility*, *correctness*, and *selectability*.

7.3.4.1 Scalability of the Reasoning

One concern with respect to service composition systems is whether they remain scalable, in our case given that the reasoner could have to consider hundreds of different services. This is especially relevant when considering remote services and knowledge sources that are available to the reasoner and are considered each time a client asks it for a computational path to a user goal. Some challenge that current reasoners are capable of processing service descriptions for applications in the context of the IoT [44] or even dismiss current reasoners as impractical for processing semantic information in pervasive computing scenarios, due to architectural and performance issues [70]. Compared to, e.g., approaches based on OWL-S, our system is very lightweight, being grounded in first-order logic. However, although other tests with reasoner-based composition already give positive indications [230], we conducted an evaluation to see how fast the reasoner we use in our system – the Euler Yap Engine (EYE) [323] – can process service descriptions when the

#services	1024	2048	4096	8192	16384	32768	65536	131072
parsing	276 ms	528 ms	1001 ms	1949 ms	3916 ms	7827 ms	17127 ms	34526 ms
reasoning	12 ms	20 ms	18 ms	68 ms	107 ms	113 ms	122 ms	228 ms
total	289 ms	548 ms	1019 ms	2018 ms	4023 ms	7940 ms	17249 ms	34754 ms

Table 7.1: Delays incurred by parsing and reasoning over many services.

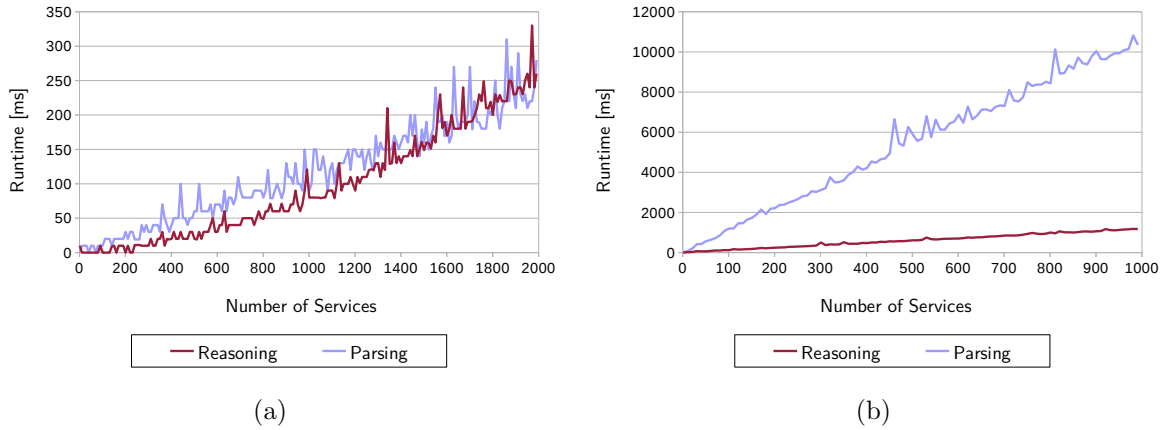


Figure 7.8: Runtime of the reasoning when considering (a) simple chains of up to 2000 services and (b) complex scenarios with up to 1000 combination options per step and a fixed composition length of 100 steps.

number of available services grows. In a first test that is described in [230], the total composition length was fixed to 32 services (which is a lot for the context we consider), and the number of services that are considered during the reasoning step was increased to up to 2^{17} . The results (Table 7.1) indicate that even for very high numbers of considered service descriptions, the reasoning time remains under a few hundred milliseconds on an average consumer computer, and thus within reasonable limits. The time required for downloading and parsing the rules, however, does significantly increase, but this effect can be mitigated by caching service descriptions locally at the reasoner.

Several tests that were carried out in the context of this thesis confirm these results: First, we let a reasoner chain up to 2000 services – our results (see Fig. 7.8(a)) show that especially the time required for reasoning stays manageable even for such large numbers of services in simple chaining scenarios. To explore the runtime behavior of our reasoner in a very complex scenario, we fixed the composition length to 100 services but varied the *number of options per chaining step* between 1 and 1000 (see Fig. 7.8(b)): again, the reasoning time is rather low with 1171 ms when chaining 100 services at 1000 options per step (i.e., 100^{1000} options in total). For both scenarios, the charts in Fig. 7.8 also record the parsing overhead – in our tests, this is much lower than the numbers shown in Table 7.1 (but remains significant) because in both our test cases, all service descriptions were contained in a single document.

7.3.4.2 Expressibility of RESTdesc

Being based on rules in the N3 format, the RESTdesc language is in principle limited in its expressiveness to implications in monotonic first-order logic [229]. This implies

that the basic system does not allow the removal of facts from the knowledge of the reasoner, which represents a particularly delicate issue when dealing with HTTP DELETE requests: “a given thing that exists cannot unexist” in the system [229]. Therefore, this approach is prone to inconsistencies, which is why we introduced a method for explicit state management to handle states in smart environments (Section 7.3.3) – this enables the system to “virtually” forget facts, by having state transitions (such as that defined by the smart thermostat in Listing 7.3) explicitly redefine the current state.

Another, related, notion that our system is not capable of expressing is time: in a world governed by first-order logic, everything that can be true is true [229]. We have not addressed this issue within the context of this thesis because it was not necessary for the scenarios we considered, due to the REST-based interaction between our endpoints. To illustrate this, consider again the smart thermostat described above where the user goal (Listing 7.4) specifies that the location “Office” should have a temperature of 23 degrees Celsius. When discussing this example, we did not go into details about the exact semantics of the `hasTemp` predicate: does it imply that the temperature as measured at the location *is* 23 degrees Celsius or merely that the setpoint of the smart thermostat has been set to that value? While the intended semantics must be specified within the `hasTemp` predicate, we argue that our system is able to deal with both interpretations, although the former implies a temporal relationship. If the semantics of `hasTemp` are defined such that the postcondition of the description exposed by the smart thermostat service implies that its location *has a certain temperature*, the thermostat should reply to the corresponding request only *after this state has been reached*. If, on the other hand, `hasTemp` merely implies that the setpoint of the thermostat is set to that value, it can immediately reply to the user request. Thus, we treat temporal constraints as dependencies on the successful execution of services that the reasoner instructs us to execute prior to another service and do not require an explicit mechanism to deal with temporal relationships in our system.¹⁰

We explored the handling of temporal constraints further in a scenario where the user goal is to move an object from one place to another. In our setup, this could only be accomplished by multiple collaborating robotic devices: a static robot would place the object on the cargo area of a mobile robot that would then transport it to a different room. This implies that the client must wait with triggering the mobile robot until the static robot notifies it that the object has indeed been placed in the cargo area – to accomplish this, we make use of the same straightforward technique as described above: the static robot consumes the client request but delivers a response only after finishing its task. Only then, the client issues its request to the mobile robot.

[207] describes several other dimensions of expressibility, some of which are relevant in the context of our system. For instance, our system does not have the capability of explicitly “modeling complex structures such as sequence, choice, concurrency and iteration” – however, thanks to the underlying semantics, it does not require these features that are indispensable in a process-driven service composition context. The reasoner

¹⁰This is a similar argument as brought forward in the discussion of dependencies between atomic interactive components of a smart device in Chapter 3: there, as well as here, we exploit features of the underlying REST architecture to move the responsibility for handling temporal/functional dependencies from the user interface to the interactive components.

furthermore automatically takes care of specifying the data flow among activities while we do not support the notion of “allocating activities to respective roles” [207] – in our system, each service (including the client interface and the reasoner) is a standalone component whose function is defined by its RESTdesc description. Finally, as mentioned before, our approach does not support exception handling, nor does it provide an explicit way of specifying what constitutes exceptional behavior. However, we believe that the HTTP status codes together with the run-time adaptability of our system are sufficient as a basic recovery mechanism for clients.

In summary, although the *explicit* expressiveness of RESTdesc does not rival that of planning languages or business process definition languages, we found that it is suitable for describing services that we encounter in WoT scenarios. The only capability that we added to the language is an explicit state handling mechanism to remove inconsistencies that could arise from state changes in smart environments. With this modification that we combined with a pragmatic approach of handling temporal dependencies, we found the system to be applicable in typical pervasive computing scenarios and used it to specify services in a home automation context, define capabilities of robotic devices with respect to the transportation of items from one location to another, and describe services provided by Web-enabled automobiles (see Chapter 8). Others have demonstrated that the RESTdesc language can also be used in the context of multimedia, mathematical expression parsing and solving, and medical imaging analysis as well as diagnosis assistance.

7.3.4.3 Correctness of Service Compositions

In principle, the reasoning component in our approach guarantees the correctness of any composite service it generates – this, however, assumes that the underlying RESTdesc documents clearly and unambiguously capture the functionality of the described services, and that the user goal correctly specifies the desired state of the smart environment using semantic concepts that are compatible to the service descriptions. While we believe that both these challenges can be overcome in limited scenarios that are under full control of a single party (or several parties that fully agree on the underlying semantic concepts), they give rise to a challenge at the heart of the Semantic Web, especially when third-party services and ontologies are incorporated in the reasoning: the issue of conflicting semantic information. This is perhaps the prime reason for many researchers to be skeptical regarding the fitness of semantic technologies in the context of real-world applications [122, 206] and to question whether these technologies are actually able to achieve the promised interoperability between services.

In the context of service composition in smart environments, conflicts in the semantic information could lead to situations where services that *should be interoperable* cannot be combined by the system and to settings where services that *should not interact* are utilized within a service mashup or, similarly, where data that is *not intended as input* for a specific service is processed by it. To give a straightforward and intuitive example, this behavior could arise if a smart thermostat is incorrectly described as taking *number* objects as input. In this case, it could happen that the reasoner constructed a composite service that takes, for instance, the *number of devices* in a smart environment as input

to the smart thermostat or, perhaps more worryingly, the *number of files* in a file system on a device in the surroundings. Subtle differences in meaning and the usage of different, incompatible, vocabularies in RESTdesc descriptions might thus give rise to false positives that entail dramatic consequences or false negatives where the reasoner fails to derive a relationship between services that might match in terms of their functionality [231]. Due to interactions between many devices in smart environments, even small errors by a user could have major consequences.

To mitigate these issues in our system, we added the option of visualizing suggested composite applications prior to executing them (see Section 7.4). Still, our system does not provide a universal remedy to these issues: there will probably always be cases in smart environments that are difficult or even impossible to solve with it. However, we believe that our system represents an advancement over comparable approaches by taking a pragmatic stance with respect to several key challenges, such as state handling and the management of dependencies between service executions. In fact, the RESTdesc language itself represents a pragmatic approach as only the minimally required knowledge is specified (thus allowing to scale to high numbers of services) and redundant constructs that characterize alternatives such as WADL are omitted: even with the proposed change to include explicit state descriptions, the total length of a RESTdesc document is typically 10 to 20 lines (excluding prefix declarations). Thus, while certainly examples can be found that cannot be solved using our approach, many do already work, and we have demonstrated the applicability of our system to real-world use cases in home automation, logistics, and the automotive domain (see Chapter 8).

In the context of smart environments, we thus view semantic technologies in the way we conceived them as a very flexible form of standardization. Certainly, standards – if honored by all relevant stakeholders – could also accomplish the use cases that we put forward in this chapter. However, while standardization can improve interoperability among standard-compliant components, it impedes or complicates the integration of elements that were out of scope at the time the standard was designed – ontologies have been shown to be more flexible with respect to adding additional concepts to deployed systems [111]. Furthermore, semantic technologies offer more freedom with respect to the description of REST endpoints and semantic goals. For instance, the RESTdesc descriptions of the services that are provided by different smart thermostats may differ, or they could use different formats of expressing the semantics of their offered functionality altogether. This heterogeneity can lead to added complexity and perhaps to mismatches. However, semantics provide the benefit of allowing decentralized decisions and can evolve faster than standards can. In our opinion, using semantics within service descriptions thus represents a lightweight approach to support new services in an evolving way – even when considering their shortcomings with respect to conflicting information, especially in large-scale scenarios.

7.3.4.4 Selectability of Service Mashups and Usability Considerations

The property of *selectability* refers to the selection of the most appropriate individual service within a service mashup among a number of functionally similar or equivalent

```

1 @prefix sreq: <http://example.org/security>.
2
3 # If the requirement is "None," this implies that "Confidential" is ok.
4 { _:secReq a sreq:None. } => { _:inferredSecReq a sreq:Confidential. }.
5
6 # If the requirement is "None," "HTTP" APIs are ok.
7 { _:secReq a sreq:None. } => { _:inferredSecReq a sreq:HTTP. }.
8
9 # If the requirement is "Confidential," "HTTPS" APIs are ok.
10 { _:secReq a sreq:Confidential. } => { _:inferredSecReq a sreq:HTTPS. }.

```

Listing 7.5: A RESTdesc description of a temperature conversion service.

options, based on non-functional characteristics such as QoS parameters [207]. Ideally, a service composition system would allow users to formulate non-functional preferences with respect to individual services or the service mashup as a whole, either directly within their goals or within accompanying input documents to the reasoner that express these desired characteristics. In this section, we show that the RESTdesc language permits the encoding of such properties within descriptions, by extending the *preconditions* of a service description with clauses that describe non-functional characteristics. In particular, we target the specification of security requirements – our goal here is to enable users to formulate basic requirements such as *confidentiality* that are then considered by the reasoner when composing the service mashup. Our proposed mechanism represents a proof of concept and we illustrate it using again the simple example of the smart thermostat and very basic rules for handling security requirements.

As a basis for our proof of concept, we define semantic rules that specify the relationships between different concepts that relate to security and privacy properties of a system and to technologies or protocols that implement these constraints and may be used by individual services. The N3 document shown in Listing 7.5 contains two such relationships between the two security requirements **None** and **Confidential** and the HTTP and HTTPS protocols, respectively (lines 7 and 10, respectively). In line 4, the document furthermore specifies a rule that expresses that *Confidential* is a “stronger” requirement than *None*. The consequence of incorporating these rules during the reasoning is that, if a semantic reasoner knows about the additional, user-defined fact `:userSecurityRequirement a sreq:None.`, it will infer three more facts that correspond to `sreq:Confidential` (by applying the rule in line 4 of Listing 7.5), `sreq:HTTP` (from line 7), and `sreq:HTTPS` (by applying the rules from lines 4 and 10). Alternatively, if we instead supply the fact that `:userSecurityRequirement a sreq:Confidential.`, the reasoner will only infer the fact that an entity of type `sreq:HTTPS` exists (from line 10). Therefore, when *describing a service*, we can now use the security constraints to express that a service uses plain HTTP without any security features by adding a corresponding precondition to its description – for the smart thermostat, this is shown in line 20 of Listing 7.6.

Given descriptions that include such preconditions and the security rules from Listing 7.5, the system will not instantiate the thermostat service as part of a service mashups whenever the client specifies a security requirement of `:userSecurityRequirement a sreq:Confidential.` in its goal, since the precondition `_:secReq a sreq:HTTP.` cannot

```

1  @prefix st: <http://purl.org/restdesc/states#>.
2  @prefix log: <http://www.w3.org/2000/10/swap/log#>.
3  @prefix dbpedia: <http://dbpedia.org/resource/>.
4  @prefix ex: <http://example.org/#>.
5  @prefix http: <http://www.w3.org/2011/http#>.
6  @prefix geonames: <http://www.geonames.org/ontology#>.
7  @prefix sreq: <http://example.org/security>.
8
9  {
10   ?newTemp a ex:Temperature;
11           ex:hasValue ?fVal;
12           ex:hasUnit "Fahrenheit".
13
14   ?thermostat a dbpedia:Thermostat;
15           geonames:locatedIn ?place.
16
17   ?state a st:State;
18           log:includes { ?place ex:hasTemp ?oldTemp. }.
19
20   _:secReq a sreq:HTTP.
21 }
22 =>
23 { (...) }.

```

Listing 7.6: A RESTdesc description of a temperature conversion service.

be satisfied. If, alternatively, the service specified a security precondition of “_:secReq a sreq:HTTPS.”, the reasoner would make use of it, since – according to the security ontology given above – this is compatible with a `sreq:Confidential` user security requirement. Note that this also works for composite mashups, i.e., it is possible for a client to define that it wishes all communication that happens as part of a mashup to happen in an `sreq:Confidential` way.

This extension gives users control over the security requirements of an application as well as other non-functional aspects that can be modeled ontologically: it permits users to specify which services from a set of functionally equivalent offerings should be selected. To give users more control about which individual services are part of a mashup that is executed on their behalf, we have also experimented with a dynamic feedback system that proposes multiple execution paths and lets the user decide which path to invoke. This system is discussed as part of Section 7.4 where we present a general approach to make our semantics-based system for the configuration of smart environments usable for end users – in particular, we show how the goal formulation step can be simplified.

7.4 Making Semantic Technologies Usable for End Users

In the previous section, we presented the RESTdesc format and an extension to RESTdesc that allows to use its description style to configure smart environments. Using this system, end users can formulate a semantic goal and submit this goal to a reasoner that will

attempt to return a series of Web service invocations to reach the goal. However, because of the underlying semantics that our system requires and since the introduction of states adds even more complexity to the goal creation, end users cannot be expected to write valid goal descriptions by themselves: the user would not only need to know about the *predicates* to use (e.g., **hasTemp**) but also about the *states ontology* itself, as well as the correct *N3 syntax* to express his goal.

In this section, we discuss two techniques to mitigate this problem by facilitating the formulation of goals for end users: First, we present an integration of the system with a visual programming language that constrains users to modeling only specific properties that are supported by their current smart environment. Second, we propose a mechanism that aims to reduce the burden on end users even further by extracting goal templates from services present in the user's environment and having users simply *select* an appropriate goal, rather than formulating it themselves.

7.4.1 A Graphical Editor for User Goals

As a first step to facilitate the process of formulating goals for end users, we integrated our system with ClickScript [305], a JavaScript-based visual programming tool. We already used ClickScript in earlier projects, when it was extended with the capability of connecting to Web services that run on smart devices using AJAX [75]. Generally, the system is used by dragging components that represent variables, control structures, or smart devices from the top bar to the editor and connecting matching inputs and outputs (see Fig. 7.9). When satisfied with the designed application, users click the “Run” button that causes ClickScript to execute it starting with initial nodes (i.e., nodes without inputs) and working its way along the mashup graph.

We have further extended the tool to enable its usage for “designing” semantic goals and for using it as an interface to a semantic reasoning service. Specifically, we have equipped ClickScript with components that represent the different predicates that are useful to describe a smart environment with entities that encapsulate the state of real-world items such as rooms (see Fig. 7.10(a)). The components available for modeling attributes of a smart environment such as an abstraction for a *Room* entity, or a thermometer that is associated with the **hasTemp** predicate can be dragged to ClickScript's editing view to use them within goal definitions.

For instance, to model different desired attributes of a room as in Fig. 7.10, the first step is to create a new room entity and to connect it to the corresponding room identifier (in this case, *Office*). Next, the user configures the desired state of this room by adding components which represent different aspects of that state: the *note* icon represents media playback, the *thermometer* icon stands for the ambient temperature (i.e., the **hasTemp** property in the goal shown in Listing 7.4), and the *alarm clock* icon is used to model ambient alarms. Finally, the user can infer the correct data types of the components' input parameters (for instance, the concrete temperature value) from the colors of the component inputs. A user's task thus only consists of dragging the desired elements to the editing view, connecting the matching input and output types, and entering the parameter values.

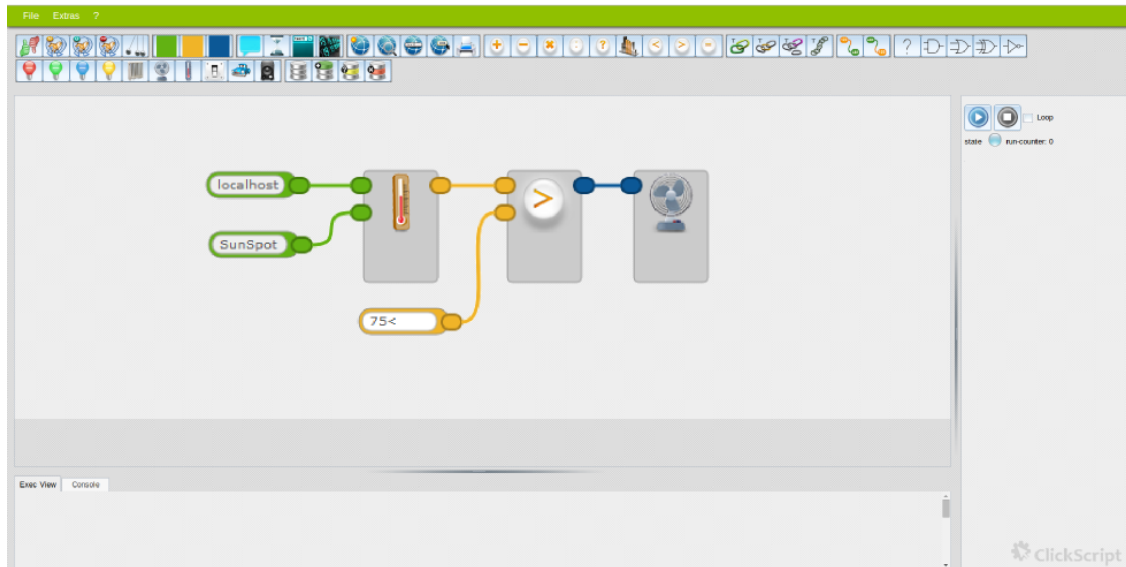


Figure 7.9: The ClickScript interface allows users to graphically create applications by connecting components from the top bar. In the example, the output of a thermometer is compared to a constant and, depending on the result of the comparison, a fan is switched on or off. The connector colors correspond to different data types *String* (green), *Number* (yellow), and *Boolean* (blue).

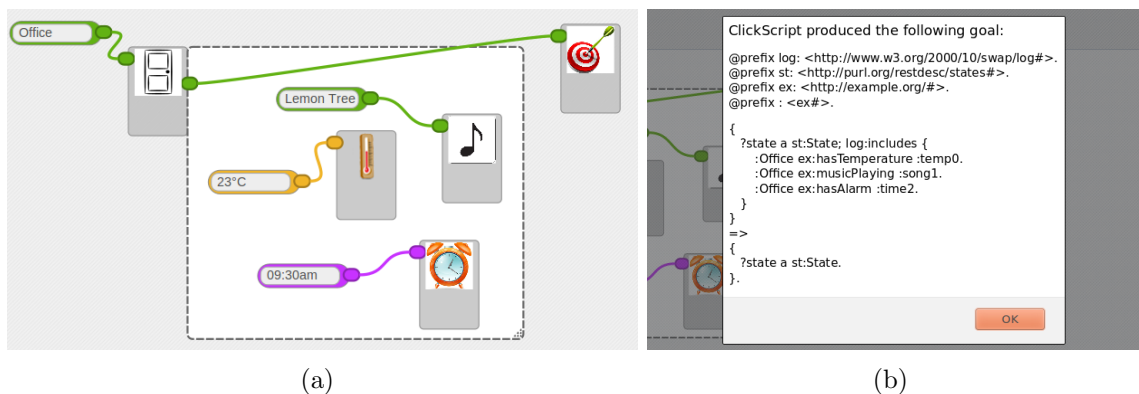


Figure 7.10: (a) With our extensions that include *Room* entities and components which represent semantic predicates such as *hasTemp*, ClickScript users can create semantic goals graphically. In this example, the user wants to configure the ambient temperature, set an alarm, and choose a currently playing song for the room “Office.” The connected “Target” component leads to ClickScript displaying the goal that corresponds to the modeled environment (b), where we omitted the definitions of the parameters *temp0*, *song1*, and *time2*.

When satisfied with the configuration of the smart environment state, the user can choose among multiple options of how the created model should be processed by ClickScript, by connecting different components to the output connector of the room entity. The user’s first option is to output the goal textually on the screen by connecting the “Target” component. This situation is shown in Fig. 7.10(a) and the goal that is displayed in N3 notation is shown in Fig. 7.10(b). Alternatively, the system can provide a human-readable description of the HTTP requests that should be executed to reach the specified state of the smart environment. In this case, ClickScript invokes the reasoner to find

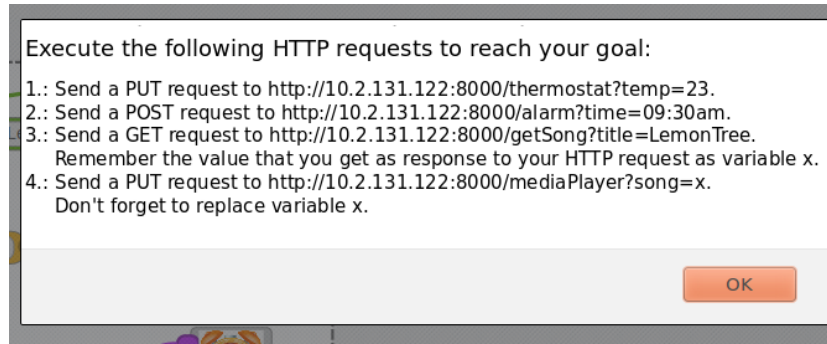


Figure 7.11: When connecting the “Question Mark” component to a room, *ClickScript* displays the necessary requests to achieve the goal in a human-readable way, using a simple verbalization heuristic.

the necessary HTTP requests, uses a verbalization service to create a human-readable representation of them, and displays this information (see Fig. 7.11). Finally, the user can choose to have ClickScript itself execute these requests and thereby directly modify his smart environment to match the modeled goal state.

Thus, to create a semantic goal using the ClickScript interface, the task of the user only consists of dragging the desired elements to the editing view, connecting the matching input and output types, and entering the parameter values. Users can then decide whether (a) the goal should be displayed, (b) the reasoner should be invoked and the necessary requests displayed in human-readable form, or (c) the necessary requests should be immediately executed by the ClickScript tool itself.

ClickScript does not only parse syntactically correct goals from modeled environments, but also constrains the user to specific ontologies that describe services in smart environments, thus mitigating the problem of conflicting semantic information discussed in Section 7.3.2. To extend the graphical interface with new predicates that can be part of modeled states, it is sufficient to specify the name of the predicate and its input type, and define a new icon that is used to represent the property. Thus, as ClickScript is based on JavaScript, only few lines of code are required to add a new component to the system. Finally, we mention that, although we did not conduct a usability evaluation, ClickScript seems to be simple to use even for people without any programming experience, according to our own and others’ experience [75, 120].

7.4.2 Automatic Creation of User Goals

Potentially, end users will never be required to formulate goals themselves, as these can be encapsulated in tailored applications on smartphones or other devices, which could also integrate further knowledge about the user’s context and his preferences. As an example, it would be perfectly feasible that experts create an application which infers favorite songs from a user’s history, creates goals to make his environment play these songs, and executes the corresponding requests without any intervention by the user. Similar applications could be created for office environments, or to support specific use cases in industrial settings.



Figure 7.12: Using image recognition software and automatic deduction of user goals from RESTdesc descriptions (mock-up): a projector acts as a proxy for service mashups that it can form part of. In this case, the projector provides the service to either display an image that is stored on the tablet device, or to display the location of the user’s car on a map.

Still, to further investigate how the configuration of smart environments could be facilitated for end users based on our system, we explored the option of deriving components of user goals at runtime in a fully automatic way, based on the available services in the user’s surroundings. Specifically, our system is able to *derive potential user goals from the postconditions* of discoverable RESTdesc documents: for instance, from the description of a smart thermostat, the system can deduce that the predicate `hasTemp` exists and that it can be used within descriptions of the state of a smart environment, and also find its data type.

Because the RESTdesc descriptions are in our case often attached to services that are provided by specific devices in a WoT context, this approach allows us to use a smart device itself *as a proxy* for all mashups that involve a goal state that corresponds to a postcondition of the description of that device. In other words, our system can take a RESTdesc document that describes a device, extract its postcondition, re-formulate the postcondition as a user goal, and ask the reasoner for all paths that lead to this goal.¹¹ By combining this train of thought with the technologies for selecting devices in smart environments that we discuss in Chapter 4), we can use a handheld device to recognize smart objects and immediately display potential actions that these can perform on behalf of the user. Fig. 7.12 illustrates this: by recognizing a projector, the system can find out that this device could display an image that is stored on the user’s tablet. Alternatively, if the system has also access to semantic descriptions of the Web-connected car of the user (see Chapter 8), it can unobtrusively propose a mashup that displays the location of that vehicle using the projector in conjunction with third-party services and the GPS sensor of the car.

¹¹The EYE reasoner was not meant to be used to find *many* paths that lead to a goal – however, its creator Jos De Roo was kind enough to extend the software when we asked for this functionality to be integrated – thank you!

7.5 Summary

In this chapter, we described two composition systems for Web services that contribute to the large body of research in that domain by suggesting approaches to enabling service composition in dynamic pervasive computing environments that heavily build on the REST architectural style. First, we described our implementation of a computational marketplace for the WoT, which is an open framework that supports the crowd-based publishing of information about Web service mashups and their usage by clients. Our marketplace was created to re-use core REST concepts to guarantee properties such as scalability, fault tolerance, and change tolerance for distributed computations. In particular, we explored the use of the HATEOAS constraint for guiding clients within service mashups. We concluded that a computational marketplace can enable clients to make locally informed decisions about the traversal in a mashup, but that additional information is necessary for fully automated usage of our system.

To facilitate the configuration of smart environments for end users by fully automating the service composition step, we propose a goal-driven approach to this challenge, where users express their needs using a graphical configuration environment. In our system, a user formulates a goal that defines the desired state of his environment which is used by a semantic reasoner to deduce the HTTP requests necessary to reach that goal. We are able to satisfy complex demands using only first-order logic, which makes this system flexible yet fast. From our perspective, using semantic technologies to deduce service mashups represents a much more flexible alternative to the process-driven composition of services: because the services are combined at runtime, the system can flexibly react to individual services becoming unavailable by finding alternative paths that also serve to reach the user's goal. Furthermore, the reasoning process could also take into account more information about the user context, or his preferences, to derive mashups that are even better suited for a concrete situation.

One major challenge in the proposed system is that the goal formulation step is hard to accomplish for end users. For this reason, we extended our system by integrating it with a graphical editor that enables users to easily create a model of the desired state of their environment and translates this model into a goal in the N3 format. This approach hides the complexity of the underlying semantics from end users and mitigates their fragility by constraining end users' leeway to the specific functionality that is offered by the graphical editor. We also explored a method of deducing user goals from the service descriptions of smart devices present in the user's environment, thereby transforming the goal formulation problem into one of merely selecting an appropriate goal.

CHAPTER 8

Case Study: Interacting with Smart Cars

A car whose telemetry data is projected into the “Cloud” and accessible for third parties via standard Web technologies represents, in our perspective, a highly interesting example of a Web-enabled smart thing – in this chapter, we discuss the application of the techniques that we propose in this thesis for interacting with smart devices to Web-enabled automobiles. The basic infrastructure necessary to make cars accessible over the Web is supplied by *CloudThink*, a platform that we developed to make vehicle data and actuation capabilities usable for clients in near real time via an intuitive REST API. Because CloudThink thus integrates cars into the Web of Things, all approaches that we present in this thesis for interacting with smart devices also apply to automobiles: we can recognize cars and interact with them, can visualize their interactions with other smart things and applications, and can semantically describe the data and services they provide to enable the automatic collaboration of vehicles with smart devices and Web services.

We begin this chapter with a discussion of the CloudThink platform and, subsequently, discuss several applications that we developed on top of the platform (Section 8.2): because the current main focus of CloudThink is to help drivers economize fuel, we first give a brief overview of an application that we developed to this avail and then discuss the integration of the platform with our interaction techniques for smart devices. Specifically, we show that our object recognition system from Chapter 4 can be combined with the embedded interaction descriptions proposed in Chapter 3 to enable drivers to directly interact with their cars. We furthermore demonstrate the applicability of our approach of semantically describing services that are provided by smart things to automobiles (see Chapter 7) – this system integrates cars with other smart devices and Web services, thus creating physical mashups that make use of data and functionality provided by vehicles. Finally, we show that our augmented-reality interface described in Chapter 5 can visualize interactions of clients with cars, thus demonstrating the applicability of the logging of HTTP requests also in this domain.

8.1 The CloudThink Platform

In the context of a collaboration between ETH, Massachusetts Institute of Technology (MIT), Singapore University of Technology and Design (SUTD), and other research institutions in the USA and the United Arab Emirates, we have – since the year 2012 – been involved with the development of a vehicle-to-cloud communication platform that we call CloudThink [239]. The core of this system consists of a low-cost hardware platform (the *CARduino*) that connects to the on-board diagnostics port of a car using the OBD-II interface¹ and a secure back-end server that stores data uploaded from cars and makes it accessible to other applications via a REST API.²

The main goal of CloudThink is to enable drivers to share data from their vehicles for processing by third-party applications while emphasizing an open, secure interface to this data. CloudThink thereby aims to create an “Appstore” for cars that hosts applications which can access and process car telemetry. The platform is distinguished from similar projects in the automotive domain in that the platform remains agnostic of the concrete type and make of vehicle and therefore enables data sharing across the boundaries of car manufacturers – use cases for the CloudThink system stretch from enhancing vehicle security (for instance, a “virtual leash” on cars, or tracking of stolen vehicles) to reducing the environmental impact of cars by helping drivers to find out how they can better economize fuel. Drivers could also use such a system to improve safety by adapting their driving style [117], and the platform could help set economic incentives to do so by enabling pay-as-you-drive insurance schemes [168, 242]. Potentially, drivers could also use CloudThink to monetize their driving data, for instance by sharing usage statistics and telemetry with the manufacturer of their car – or provide the same data to non-profit organizations to achieve better transparency of how specific vehicles fare in realistic situations, for instance with respect to their fuel consumption or maintenance effort. Another highly interesting application domain is using real driving data for the improvement of vehicle characteristics, for instance determining the optimal battery size for electric cars [212]. On a larger scale, real-time feedback from vehicles could help improve road quality – and safety – by enabling road operators to quickly react to problems that range from potholes to developing traffic jams [195, 209].

The main components of the CloudThink system are the CARduino hardware, the data server that stores data uploaded from individual cars in a local database, and the gateway server that enables third parties to access the stored data in a uniform way via a REST API (see Fig. 8.1). While our partners at MIT are leading the hardware development (see Section 8.1.1), our main responsibility within the CloudThink project is the implementation and maintenance of the back-end infrastructure (see Section 8.1.2). We also created applications that provide better access to data stored on the platform in the form of a “virtual dashboard” that could be used, for instance, by fleet managers, and used telemetry data from CloudThink to create an application that gives drivers more insight into their own driving behavior – and clues as to how they could improve their own fuel economy (see Section 8.2).

¹This is a standard interface that every car commissioned after the year 1996 must support.

²See <http://api.cloud-think.com>

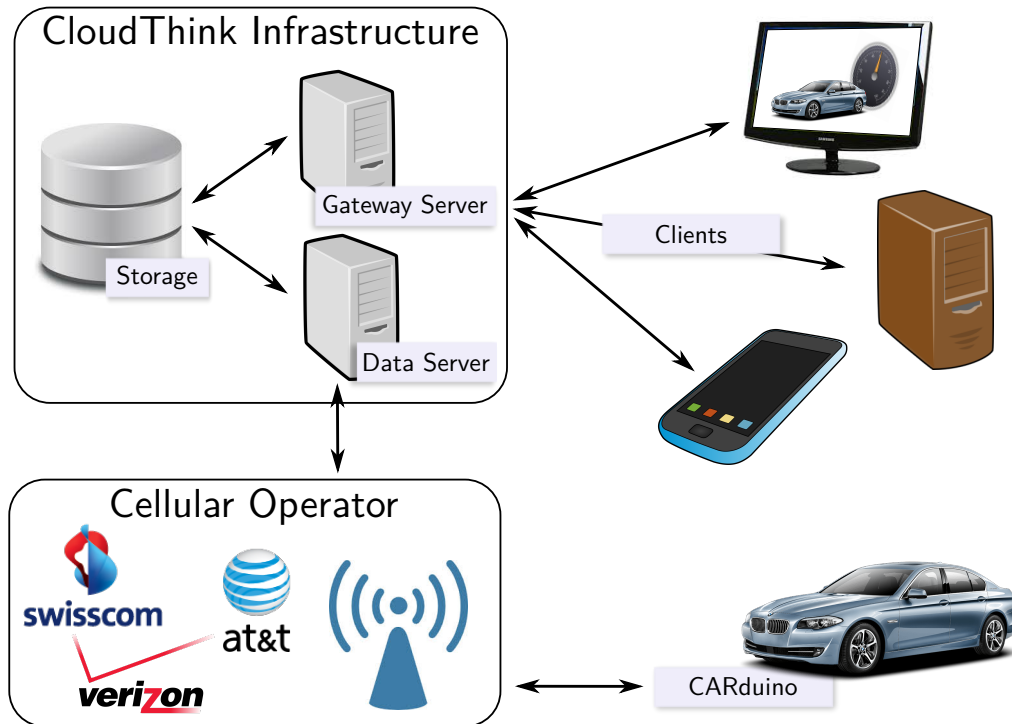


Figure 8.1: Overview of the CloudThink architecture. The CARduino hardware is deployed on the car and communicates with the CloudThink back end via GSM. In the back end, the data server parses and filters the data and stores it in a database. The gateway server provides a REST API for clients to conveniently access the data.

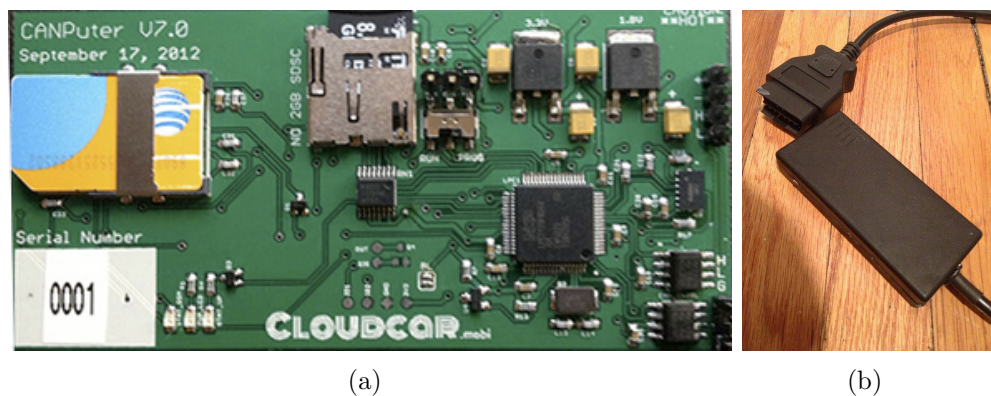


Figure 8.2: (a) The CARduino hardware (ca. 9.3cm x 4.4cm x 1.5cm). One can recognize the SIM card and the SD card that is used for data buffering. The GPS and GSM antennas are attached to the bottom of the module. The “CloudCar” silkscreen refers to the former name of the CloudThink project. (b) The CARduino hardware shown in its 3D-printed casing and the OBD-II connector cable.

8.1.1 CARduino Hardware

Our CARduino hardware³ (see Fig. 8.2) is responsible for transmitting information from the vehicle to the CloudThink back-end infrastructure, where this data is stored in a SQL-based database. All information is sent via GSM connections and encoded in a custom message format that is tuned to minimize the volume of the transmitted data, thereby helping to keep communication cost low. Each message that is sent to the back end consists of the recording timestamp, the type of the transmitted data, the data itself, and a checksum. Messages are received by the CloudThink data server that parses and checks each message before storing the contained data in the database.

The CARduino carries an on-board accelerometer and a GPS receiver, and can be configured to listen on a car's OBD-II interface for OBD messages that are tagged with specified OBD-II Parameter Identifiers (PIDs). This allows us to capture information about most internal processes of a vehicle, for instance basic driving data (e.g., speed, heading, etc.) and information related to motor management (e.g., motor revolutions per minute, mass air-flow, coolant temperature, etc.). In particular, we are able to gather all data required for calculating an approximation of the instantaneous fuel consumption of a car, which can be computed from the current vehicle speed and the mass air-flow rate [321]. When the CARduino is unable to reach our back end, for instance due to poor network connectivity, it uses an on-board memory unit for buffering and transmits the contents of that buffer at the end of the trip. Finally, our hardware can also write to the car's OBD interface, thereby enabling it to actuate vehicle functions – we have successfully used it to lock and unlock cars, and to control wipers and power windows.

8.1.2 Back-end Infrastructure

After being stored in the CloudThink database by the data server, the vehicle information is processed by a synchronization script that aligns all received data points to common 5-second time windows using linear interpolation and, as part of this process, copies it to a different database. This script is also responsible for calculating several metrics that are derived from the raw data – such as the instantaneous fuel consumption, and distance driven – and persisting these values together with the other synchronized data points.

It is the responsibility of the CloudThink gateway server to give client applications (which can be mobile applications, third-party servers, and Web browsers) access to the stored, synchronized data via a secure REST interface (SSL/TLS and HTTP Basic authentication). All data is provided in the JSON and XML formats for machine clients, and can be browsed and visualized using the HTML interface of the gateway server (see Fig. 8.3). The server provides a layer of abstraction on top of the OBD PIDs that are used internally by our system and enables clients to request data about attributes such as “RPM,” “consumption,” and “speed” – information about which types of data a specific user is allowed to access on a specific car can also easily be loaded from the interface. Additionally, the gateway server enables clients to send actuation commands to vehicles.

³The CARduino hardware is open-source. At the moment, its only manufacturer is CarKnow Ltd., a company that has been incorporated by one of our partners in the CloudThink project.

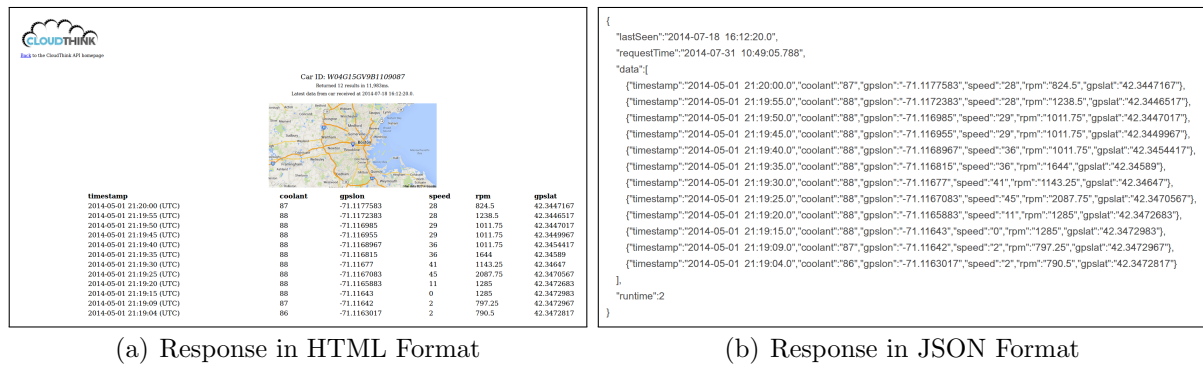


Figure 8.3: Example responses of the CloudThink API to a request for a minute of car telemetry (GPS coordinates, coolant temperature, speed, and revolutions per minute).

Another responsibility of the gateway server is to enforce that only authorized users may access data stored in the CloudThink back end. For this, it makes use of another database that stores, for each registered user and application, the data fields that this user may access (e.g., GPS location, acceleration, etc.) as well as, for each vehicle, the users that are allowed to access data from this vehicle and whether they also hold actuation rights. The gateway server also keeps track of which user accesses which stored data item for billing purposes and to be able to inform each driver about who requests what kind of information about their cars. Finally, the gateway server features a public interface that enables developers of client applications to test their ideas and software prior to registering their applications as data users on the CloudThink platform: we provide a complete set of real data from a single car for this purpose, and have a car simulation application that constantly replays a set of sample data, for the purpose of supporting the testing of applications that make use of the near-real-time data stored on the CloudThink platform. The capabilities of the gateway server are detailed in its online REST API description, along with example queries. To further support developers in interfacing with the platform, we provide sample code for multiple platforms and programming languages.

8.2 CloudThink Applications

Essentially, CloudThink aims to provide convenient access to vehicle data for third-party applications – how exactly these process the provided data to create advanced services for drivers and third parties is left to the application developers – as mentioned in the introduction to this chapter, we imagine that CloudThink could enable applications in a broad range of domains. In this section, we introduce several use cases that were implemented on top of the platform, including a virtual dashboard (see Section 8.2.1) and a fuel economy assistant (see Section 8.2.2). Primarily, however, we discuss the interaction with Web-connected cars using the techniques we propose in this thesis: the automatic generation of user interfaces and its combination with object recognition technologies (see Section 8.2.3), the visualization of interactions with Web-enabled cars (see Section 8.2.4), and the semantic annotation of services provided by smart vehicles to enable the integration of their functionality within physical mashups (see Section 8.2.5).

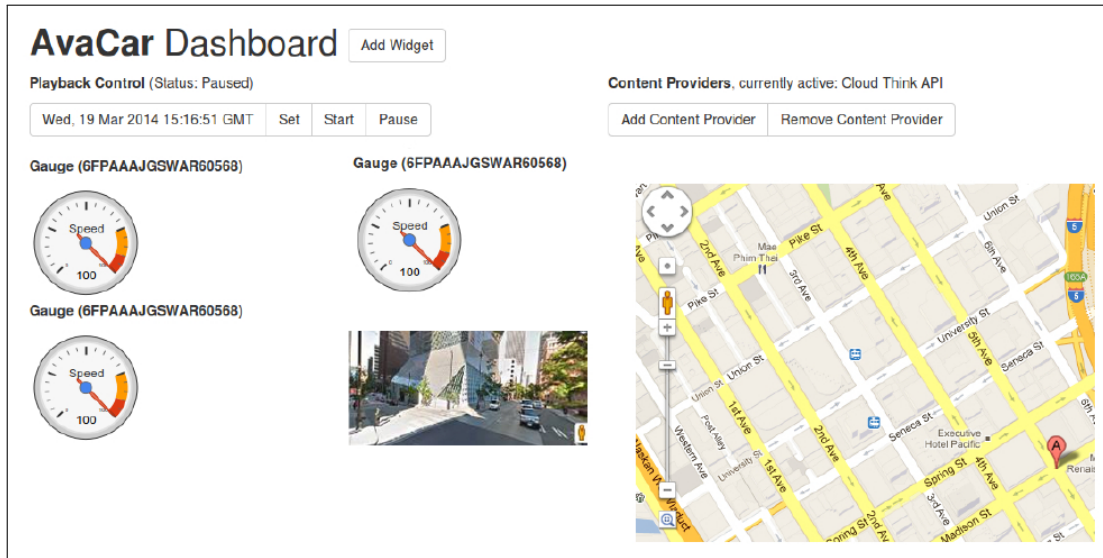


Figure 8.4: A virtual dashboard that displays data from different sensors of a Web-enabled car. The current location of the vehicle is shown on a map and in the form of a Google StreetView image.

8.2.1 Virtual Dashboard

One of the first applications that were developed to make use of the data made available via the CloudThink platform is a virtual dashboard that can visualize the current status of a vehicle using a number of user-configurable widgets (see Fig. 8.4). The dashboard is written in TypeScript, a programming language that is compiled to JavaScript and facilitates the application of object-oriented programming techniques. Its basis are several predefined widgets that hold a description of which sensors they depend on and subscribe to a central entity that polls the CloudThink API and informs each widget about updates to its relevant sensor values in a publish/subscribe paradigm.

While this application is of limited use while driving a vehicle and thus having access to its physical dashboard, we believe that virtual car dashboards can be valuable to fleet managers. The developed interface could for instance enable them to obtain a near-real-time overview of their managed vehicles that includes data about the latest instantaneous fuel consumption values, tank fill level, and total distance driven. The interface can also make use of remote Web services, for instance using the Google Maps API to display the current locations of vehicles.

8.2.2 Improving Drivers' Fuel Economy

One way that the CloudThink platform could help to reduce the environmental impact of a vehicle is by enabling to give “eco-driving” feedback to the driver. We implemented a smartphone application that makes use of vehicle telemetry data accessed via CloudThink and allows drivers to compare their fuel efficiency on different routes that they use frequently. To accomplish this, we segment an individual vehicle’s data that we obtain from the platform into individual trips and group similar trips together – from these groups, we extract the most frequently driven routes (such as the daily commute to and from work)

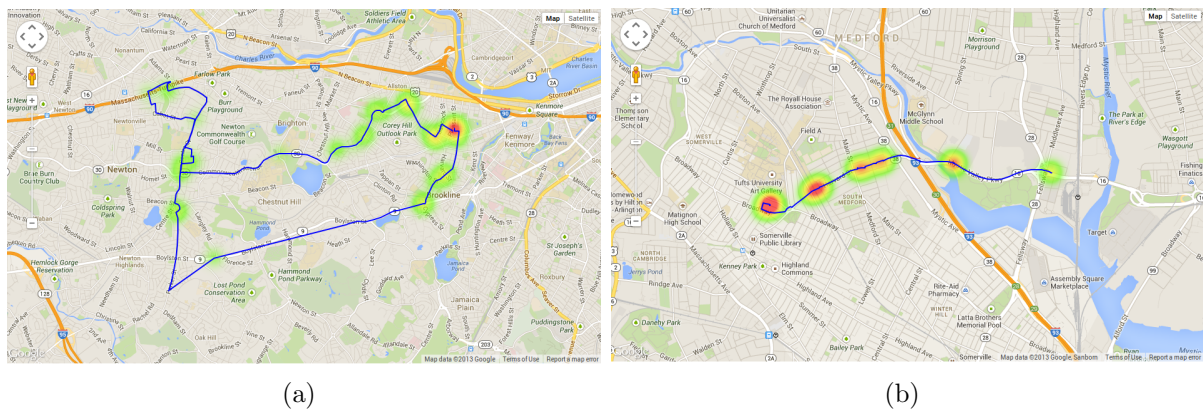


Figure 8.5: CloudThink enables the creation of heatmap-overlays that highlight places with lowest fuel efficiency such as frequently congested roads and parking lots, in this case for two areas in the Boston metropolitan area.

and compute the average fuel consumption over all trips on these routes. After these steps, we can rank individual trips on the same route with respect to fuel efficiency and give feedback to drivers about their most recent trip on that route, or let them compare a recent trip to their “reference trip,” i.e., that with the lowest fuel consumption. Drivers can also use our application to compare the environmental impact of a trip with that of alternative means of transportation, such as walking, cycling, or using public transport.

Our trip segmentation uses a dwell-time based heuristic that considers consecutive readings as a trip whenever they are not separated by a gap of at least 30 minutes. We use a higher threshold than other studies [214] to account for smaller gaps that are due to missing data which can occur, for instance, in areas without network coverage such as tunnels. Using this method, our application identified 150 distinct trips from a vehicle’s telemetry data over a timeframe of six months (128612 data samples, a total of about 9000 km driven) with a mean duration of about 32 minutes.

For the grouping of trips, we perform similarity-based clustering: the main challenge here is that the recorded GPS location traces of two trips can be very different, even though both trips correspond to the same route. This is due to variations in trip parameters such as traffic conditions, velocities, or waiting times at traffic lights: if a driver must wait at a traffic light in one trip but not in another, the results from a one-to-one matching of GPS readings can be misleading. To overcome this challenge, our route matching algorithm takes multiple observations within a specified search radius into account when matching GPS locations in two trips [233]. The algorithm then finds the longest common subsequence of matched GPS locations of two trips. For our data, we identified 30 distinct routes which comprise five trips each, on average. Note that it is intentional that trips are clustered using the described method and not via the more robust approach of matching trips’ starting and end points: we intend to do the clustering based on the exact path taken by the vehicle in each trip because our goal is not to find the most fuel-efficient *route* from one place to another (this is a closely related problem) but rather help drivers reduce the fuel consumption of their vehicle *on a specific route*.

By applying the segmentation and clustering algorithms to data obtained from Cloud-

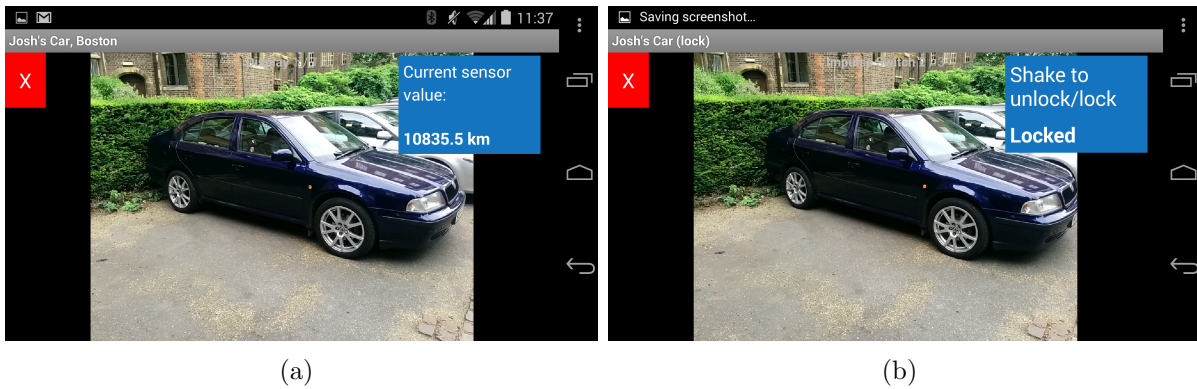


Figure 8.6: With the help of the CloudThink API, a smartphone application can display values from sensors of a recognized vehicle (a) and provide user interfaces to control its actuators (b).

Think, we can immediately compute the fuel consumption of the vehicle for individual trips, and compare trips in the same cluster. We can also generate fuel consumption heatmaps that can be overlaid on maps to highlight places where fuel efficiency is lowest (see Fig. 8.5). When applied to our data, these heatmaps indicate higher average fuel consumption on parking lots as well as hilly and frequently congested roads.

8.2.3 Interacting with Smart Cars

As an example of how the techniques for interacting with smart devices developed in this thesis can be applied to the interaction with Web-enabled automobiles, we present the usage of our approach of visually recognizing objects and automatically rendering intuitive user interfaces in conjunction with connected vehicles as a proof of concept. Our approach of describing the high-level semantics of interactions with Web-enabled devices that is discussed in Chapter 3 can be applied in the context of generating user interfaces for the sensors and actuators of Web-enabled cars without requiring any changes to the system (see Fig. 8.6) – this further emphasizes the broad applicability of our interaction description language to use cases in the ubiquitous computing domain. However, applying the visual device selection system that we propose in Chapter 4 in this context (see Fig. 8.7(a)) gives rise to the challenge of the *visual similarity* of distinct vehicles.

Visual Similarity of Different Vehicles Being based on visual object recognition technologies, our systems cannot differentiate between different cars that look alike, with the exception of cases where unique features of the car are visible in the camera frame (and training images). While our approach is thus able to find and match visual features of vehicles, these are often located on the rims or the number plate of a vehicle, on custom stickers, “inside” the car, or on its reflective surfaces (see Fig. 8.7(b)), which makes the robustness of our system heavily dependent on the current lighting conditions. Furthermore, especially in the case of “plain vanilla” cars without any distinct features, our software is not able to differentiate between cars of the same make, type, and color. Similar to humans, who often arbitrate between different cars that look alike by remembering their vehicle’s parking location, we propose to partially overcome this challenge by using

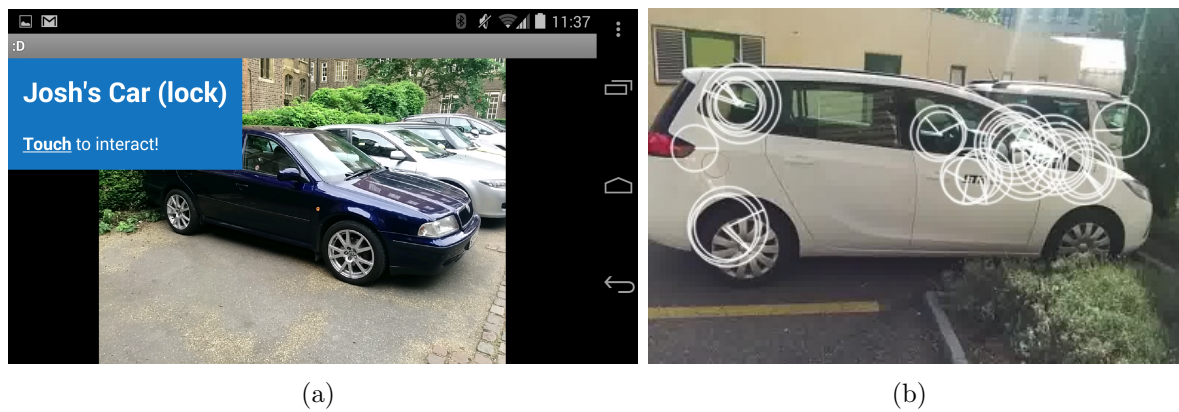


Figure 8.7: (a) Our smart things interaction software recognizes a car using visual features. (b) For cars, our visual object recognition software falls back to the few distinctive features of a vehicle, for instance on its rims, stickers, and the number plate. Unfortunately, many of the other features are located “inside” the car.

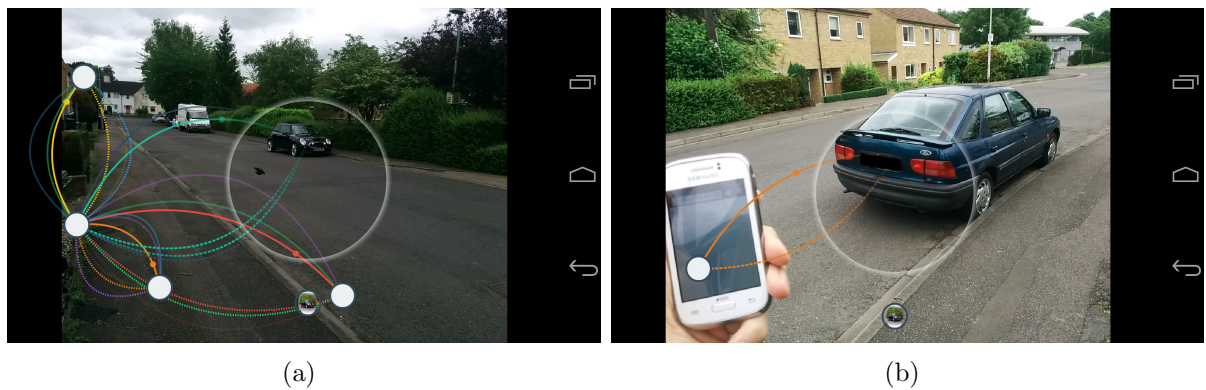


Figure 8.8: Our application visualizes interactions with a Web-enabled automobile that are brokered by the CloudThink platform using a visual object recognition algorithm and an augmented reality overlay on a handheld device. (a) Interactions of a client with a vehicle (and other endpoints). (b) Interaction of a handheld device with a vehicle.

the GPS location of a car (that can be obtained from the CloudThink API) in conjunction with the location of the user interface device as an additional context parameter to uniquely identify a vehicle (see also Section 9.3 about future work for a more general discussion of this method). However, while this technique can indeed help to achieve stable selection results, it introduces another challenge: the (remote or local) application that helps the user interface device arbitrate between different cars requires access to the locations of cars that are registered to the CloudThink platform or, as a minimum, an interface to send geographically bounded queries for a car (i.e., “Is car C currently located within 100 meters of the GPS coordinates (Lat, Lon)”).

8.2.4 Monitoring Car Interactions

Smart, Web-enabled vehicles also represent an interesting use case for our visualization system for smart device interactions. In this context, the ability to monitor interactions is

relevant to protect drivers from attacks on their privacy such as location profiling or the misuse of private data by authorities [49]. To demonstrate the applicability of our system in this domain, we integrated the active logging software of our interactions monitoring system described in Chapter 5 with the CloudThink back end: using the `javaagent` JVM parameter when restarting the CloudThink gateway server, the integration itself proved to be straightforward and could be done within a few minutes. The combined system allows users to monitor interactions of clients with their vehicles using the Web interface of our visualization tool as well as displaying these interactions as a live overlay on handheld devices (see Fig. 8.8).

8.2.5 Functional Semantic Metadata for Smart Cars

Apart from applying our approaches to facilitating the *direct* interaction with smart devices to automobiles and enabling drivers to monitor interactions of their vehicles with other clients, we have used functional semantic metadata – in the form described in Chapter 7 – to describe the functionality of the CloudThink API with respect to providing access to vehicle telemetry data of individual cars, where each car is uniquely identified using its Vehicle Identification Number (VIN). As an example, Listing 8.1 shows the semantic description of the API that enables clients to obtain the GPS latitude of the current location of a vehicle given its VIN – the CloudThink API contains a similar description for each parameter that can be accessed about its connected cars (e.g., their velocity, coolant temperature, and revolutions per minute).

To demonstrate that our system is capable of creating composite services on top of data produced by vehicles and published via CloudThink, we used its semantic descriptions together with those of a service able of displaying arbitrary images on a screen and

```

1  @prefix : <ex#>.
2  @prefix ex: <http://example.org/#>.
3  @prefix http: <http://www.w3.org/2011/http#>.
4  @prefix dbpedia: <http://dbpedia.org/resource/>.
5
6  {
7    ?car a dbpedia:Car;
8        ex:hasVin ?vin.
9  }
10 =>
11 {
12   _:request http:methodName "GET";
13             http:requestURI ("https://api.cloud-think.com/
14                             data/"?vin"?param=gpslat");
15             http:resp [ http:body ?gpslat ].
16
17   ?gpslat a dbpedia:Latitude;
18           ex:derivedFrom ?car.
19 }.

```

Listing 8.1: Part of the RESTdesc description of the CloudThink API that describes how the current GPS latitude of a car can be obtained given its VIN.



Figure 8.9: By publishing semantic metadata that describes its interfaces, functionality of CloudThink API (i.e., access to sensor values and actuators of connected vehicles) can be flexibly integrated with other local and remote Web services. In this example, a user instructs an interface device (in this case, a smartphone) to display the location of his car on a map that is shown on a nearby computer screen and a reasoner deduces the necessary HTTP requests.

semantically annotated two mapping applications (Google Maps and OpenStreetMaps). We also formulated a goal that expresses that we want to obtain an image object that is also a map and is derived from a specific car, and that a screen that is situated at the current location of the user should be displaying this image – this leads the reasoner to propose that it could display the current location of a car on a nearby screen, by mashing up the CloudThink back end with either Google Maps or OpenStreetMaps and the screen API (see Fig. 8.9). The reasoner successfully sends the appropriate requests to the client which can execute them and thereby achieve its desired goal. Furthermore, when we blocked access to one of the two mapping services, the mashup seamlessly switched to using the other, thus demonstrating the flexibility of our approach of using functional semantic metadata for service composition.

8.3 Summary

The CloudThink platform, a joint effort of ETH and several international partner institutions, has been created to make automobiles “first-class citizens” of the Web and enable third-party applications to easily access and process vehicle data while providing advanced services to drivers. We and others have created several such applications on top of this platform, in particular in an effort to help drivers increase their vehicle’s fuel efficiency. In this chapter, we briefly described one such application as well as a virtual dashboard that allows individuals to remotely access and monitor near-real-time telemetry data of automobiles, which we believe could be helpful in the context of administering vehicle fleets. We also showed that our approaches for interacting with devices in Web-enabled smart environments can be applied to vehicles that are connected to the CloudThink platform: using our systems, it is possible to directly interact with sensors and actuators of connected automobiles, monitor the communication of vehicles and clients in real time, and to seamlessly use functionality that is provided by cars within physical mashups.

The goal of this thesis was to explore how the interaction with Web-based smart environments can be facilitated for people. In the thesis, we discussed several ways of how this can be accomplished for both, the direct interaction of individuals with *individual smart devices* and the management and configuration of *entire smart environments*.

9.1 Interacting with Individual Smart Things

To support users who wish to interact with a smart thing, they must first be enabled to initiate the interaction by selecting a device suitable to their needs. We propose to make use of current visual object recognition methods that can be deployed on handheld or wearable devices such as smartphones, smartwatches, or smartglasses (see Chapter 4). For the scenarios that we considered from the ubiquitous computing domain – smart homes, workplaces, and cars – we found our approach that combines a Bag of visual Words with linear SVMs and FAST/ORB feature detection/description to be appropriate since it allowed to recognize smart devices accurately yet fast. The proposed system thus represents an intuitive and feasible way of initiating interactions with smart things in scenarios with a limited number of devices. The biggest advantages of our method are that the training can be done using only snapshots of the devices to interact with and that our approach requires no fiducial markers to be attached to them.

To supply users with an appropriate and intuitive user interface to interact with a selected smart thing, we propose to embed a description of the high-level semantics of interactions with that device within its Web representation (see Chapter 3). The advantages of this approach are manifold: interfaces can be generated for multiple interaction modalities, including gestures, speech, and interaction via physical sensors; the description itself is simple to generate even for end users, which we demonstrated in an extensive study among approximately 800 participants without any special training; the capabilities of the interaction device can be taken into account when rendering a user interface; finally, interactive components of devices and user interfaces are decoupled, meaning that the smart things do not need to be aware of the types of devices that control them.

Together, these approaches enable the intuitive and convenient direct interaction with smart devices – our approach achieved more than 75 points on the System Usability Scale which is a very high value for a prototype implementation.¹ The proposed systems can be deployed on a multitude of user interface devices that range from powerful smartphones and tablets and wearables such as smartglasses and smartwatches to simple Web-enabled switches or knobs. In particular, we have demonstrated an approach called “user interface beaming” that allows to combine the straightforward and intuitive selection of devices using smartglasses with the convenient interaction with them via a smartwatch.

9.2 Configuring and Managing Smart Environments

A necessary precondition for enabling users to interact with collections of smart devices is that these can be easily discovered, even in densely populated smart environments. To allow this, we implemented a management infrastructure that is optimized for providing discovery and look-up services for large numbers of devices in extensive settings (see Chapter 6): we guarantee its scalability by structuring its individual nodes hierarchically by logical location identifiers and letting each node administer only a single location, thereby keeping the number of devices per node manageable. In the thesis, we show that the adoption of widely used protocols and patterns from the Web in the design of our system is beneficial not only for facilitating user interaction with it but also to balance the overall load on the infrastructure and achieve high throughput when processing large numbers of look-up requests.

To enable users to control an entire smart environment that is populated by heterogeneous devices and services, we propose a flexible yet lightweight way of automatically combining Web services to achieve a specified user goal (see Chapter 7). To accomplish this, our smart devices advertise their high-level functionality using a language called RESTdesc that merges such data with information about the HTTP request necessary to invoke the device functions. Given a number of RESTdesc documents and a user goal, a semantic reasoner can combine functionality across smart devices and supply the user with all information required to execute the created physical mashup. In the thesis, we propose an extension to the RESTdesc language that makes it usable in smart environments where physical entities carry a state and, therefore, the first-order-logic-based reasoning cannot operate without modification – this enables users to configure their smart environment according to their needs by merely formulating a goal state and executing the HTTP requests suggested by the reasoner. To make the goal formulation step simple for users, we present an integration of our system with a graphical programming language called ClickScript that can create semantic goals from a graphical representation of the desired state, display the necessary HTTP requests in a human-readable way, and execute them on behalf of the user. The main advantage of our proposed service composition system are that it is highly flexible and fault-tolerant because mashups are created at runtime and, therefore, can adapt by circumventing services that become unavailable while considering newly discovered ones. Furthermore, the brittleness and potential inconsistencies of the

¹Note that this study was carried out among participants of a ubiquitous computing conference.

underlying semantics are mitigated by the ClickScript user frontend that guarantees the correct syntax and constrains users to predicates and namespaces that occur within their current smart environment.

We furthermore propose a novel method to increase the degree of information and control of users about interactions between devices in their smart environment and with remote services. We show that it is possible to inform users about such interactions by combining a logging back end with our visual object recognition system and an augmented reality overlay to visualize communications between devices in real time (see Chapter 5). Finally, we demonstrate the applicability of our proposed systems to interact with individual devices and service mashups in the context of the CloudThink platform that makes data and functionality provided by automobiles directly accessible for clients via the World Wide Web (see Chapter 8).

In conclusion, we have discussed different approaches that we developed for users to better interact with individual smart things and entire smart environments populated by Web-connected devices. Novel interaction mechanisms such as those proposed in this thesis are, in our opinion, crucial to give users control about everyday processes in an increasingly computerized world – at their homes, workplaces, and in public spaces.

9.3 Future Work

We suggest several ideas for the extension of the approaches and implemented prototypes that are presented in this thesis and briefly discuss remaining challenges that should be addressed in the future. Our suggestions refer to the usage of more information about the context of an interface device to *improve the object recognition* as well as the *user interface generation* and the importance of *(indoor) location information* as part of this context and its structuring. Furthermore, we suggest that also *interaction devices* could advertise their interaction semantics, briefly discuss the merits of a potential *automatic classification of device interactions*, and suggest two avenues for further research within the semantic service composition domain: exploiting the RESTdesc description style with respect to *global non-functional characteristics* of mashups and exploring how our system could support *multi-user scenarios*.

Context-based Device Recognition In Chapter 4, we propose the usage of visual object recognition techniques to enable users to select devices to interact with in smart environments. To accomplish that in our prototypes, we use state-of-the-art methods regarding the feature detection and description as well as the classification itself. We have shown that we can robustly differentiate about a dozen devices based on these technologies, which we deem sufficient for our scenarios – however, we expect the performance of our proposed system to decrease when more devices are considered. To overcome this challenge, we propose to make better use of aspects of the context of the device that runs the recognition algorithm other than visual features in its camera feed. Regarding such a broadening of a scene understanding algorithm beyond the visual computing domain, we particularly suggest to make use of the (indoor) location of the device and also consider

sensors such as its microphone, gyroscope, and accelerometer. The fusion of sensor data from a variety of sensors of a smartphone has already been successfully applied within the domain of activity recognition [244] – we believe that a similar system could help to increase the performance and scale of object recognition techniques as well.

Context-sensitive User Interfaces Regarding the provisioning of appropriate user interfaces (see Chapter 3), we believe that it would be interesting to explore how context-sensitive interfaces could be better supported by tailoring interaction patterns to the current situation of their user. For instance, human users often wish to interact differently with a device in their immediate vicinity than when controlling the device from a remote location. Most users probably prefer gradual/relative interaction primitives over absolute interactors to interact with devices which they can physically observe (e.g., dimming the lights in the room they currently are in) while they will prefer the latter for remote control (e.g., remote-controlling the lighting in their home from abroad). Similarly, users might prefer to interact differently with a smart device depending on other activities they are currently engaged in (e.g., driving a car, or a bicycle). Information about the context of users, and in particular their location relative to an interactive component, could thus be considered not only for improving the device selection, but also for rendering more suitable user interfaces.

Standard Logical Place Identifiers Enabling a device to determine its current location – an important characteristic of the device context – requires a method of achieving robust indoor localization. After more than a decade of intense research into this delicate topic, we are very pleased to finally also see major companies investing in real-world deployments of such systems in the last few months: Apple, a technology firm, has recently launched its iBeacon service to track shoppers at its own and others' stores [263] and Google, an Internet company, has been collecting indoor mapping data since the year 2011 [303]. If indoor localization systems are indeed widely adopted, one main concern will be to establish standard naming conventions for logical places such as rooms, shops, and buildings. Within our implementation of a management infrastructure for smart devices that also makes use of logical place identifiers (see Chapter 6), this information is statically assigned from a central registry – we do not know, however, whether such a system would be applicable within real-world scenarios, and especially when people disagree about what name to assign to a specific place.

From Interaction Descriptions to Interactor Descriptions Our approach of modeling interaction semantics to generate appropriate user interfaces could be applied not only to interactive components but also to interactors such as physical buttons or software primitives. For instance, a knob could embed a description indicating that it is usable for controlling any interactive component that has the type `set value` (from our taxonomy proposed in Chapter 3). Given this information, one can envision the user-driven or even automatic matching of interactive components and interactors in flexible smart environments. If desired by users, physical interactors such as switches could easily be

assigned to control any device that has appropriate interaction semantics and would not be anymore limited to controlling statically assigned interactive components.

Automatic Classification of Device Interactions Regarding the visualization of device interactions that we presented in Chapter 5, it would be highly interesting to combine our approach with features of current communication analysis software that is used within several commercial network management systems. This would allow the system to learn which interactions in a smart environment are considered normal and to notify the user about unusual events. The technique could also be combined with our approach of letting users set firewall rules using the augmented reality interaction visualization interface, thus allowing them to configure the system to automatically prevent unusual network messages.

Global Non-functional Characteristics in Service Composition In Chapter 7, we proposed to use the preconditions of RESTdesc descriptions for controlling the non-functional properties of a service mashup, in particular to express simple security guarantees of individual services. The potential of extensions to the preconditions and postconditions of RESTdesc should be further explored to enable clients to better manage the characteristics of generated service compositions, for instance regarding QoS properties. Specifically, we believe that it is feasible to exploit the RESTdesc style to guide the service composition regarding global non-functional properties of mashups, for instance their total expected latency or cost, and their end-to-end security properties.

Service Composition in Multi-user Environments Finally, we suggest that it should be investigated how goal-based service composition systems could support multiple users in a smart environment [121]. While we believe that majority- or consensus-based systems should be feasible in multi-user environments, we have not implemented any multi-user support. The semantic reasoning that we propose in Chapter 7 is not aware of whether it is used by one or multiple users and our system tenaciously executes requests in the order of their arrival, thus implementing a “last-request-decides” metric.

- [1] M. Agrawal, K. Konolige, and M. R. Blas. CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching. In *Proceedings of the 10th European Conference on Computer Vision (ECCV)*, volume 5305 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2008.
- [2] A. G. Aguilar. Exploring Physical Mashups on Mobile Devices. Master’s thesis, ETH Zurich, Switzerland, 2010.
- [3] A. Alahi, R. Ortiz, and P. Vandergheynst. FREAK: Fast Retina Keypoint. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 510–517. IEEE Computer Society, 2012.
- [4] R. Alarcón and E. Wilde. Linking Data from RESTful Services. In *Proceedings of the WWW2010 Workshop on Linked Data on the Web (LDOW)*, volume 628 of *CEUR Workshop Proceedings*, 2010.
- [5] R. Alarcón, E. Wilde, and J. Bellido. Hypermedia-Driven RESTful Service Composition. In E. Maximilien, G. Rossi, S.-T. Yuan, H. Ludwig, and M. Fantinato, editors, *Service-Oriented Computing*, volume 6568 of *Lecture Notes in Computer Science*, pages 111–120. Springer, 2011.
- [6] E. Avilés-López and J. A. García-Macías. Mashing up the Internet of Things: a framework for smart environments. *EURASIP Journal on Wireless Communications and Networking*, 2012.
- [7] K. Baïna, B. Benatallah, F. Casati, and F. Toumani. Model-Driven Web Service Development. In *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 290–306, 2004.
- [8] R. Ball, G. A. Fink, and C. North. Home-centric Visualization of Network Traffic for Security Administration. In *Proceedings of the 2004 ACM Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC)*, pages 55–64. ACM, 2004.
- [9] S. Bandyopadhyay, M. Sengupta, S. Maiti, and S. Dutta. A Survey of Middleware for Internet of Things. In A. Özcan, J. Zizka, and D. Nagamalai, editors, *Recent Trends*

- in *Wireless and Mobile Networks*, volume 162 of *Communications in Computer and Information Science*, pages 288–296. Springer, 2011.
- [10] S. Barker, A. Mishra, D. Irwin, P. Shenoy, and J. Albrecht. SmartCap: Flattening Peak Electricity Demand in Smart Homes. In *Proceedings of the 10th IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 67–75. IEEE Computer Society, 2012.
 - [11] M. Bauer, C. Becker, and K. Rothermel. Location Models from the Perspective of Context-Aware Applications and Mobile Ad Hoc Networks. *Personal and Ubiquitous Computing*, 6(5/6):322–328, 2002.
 - [12] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, 2008.
 - [13] M. Beaudouin-Lafon. Designing Interaction, not Interfaces. In M. F. Costabile, editor, *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI)*, pages 15–22. ACM, 2004.
 - [14] C. Beckel, L. Sadamori, and S. Santini. Automatic Socio-Economic Classification of Households Using Electricity Consumption Data. In *Proceedings of the 4th International Conference on Future Energy Systems (ACM e-Energy)*, pages 75–86. ACM, 2013.
 - [15] M. Beigl, A. Schmidt, M. Lauff, and H.-W. Gellersen. The UbiCompBrowser. In C. Stephanidis and A. Waern, editors, *Proceedings of the 4th ERCIM Workshop on User Interfaces for All (UI4All)*, pages 51–86, 1998.
 - [16] M. Beigl, T. Zimmer, and C. Decker. A Location Model for Communicating and Processing of Context. *Personal and Ubiquitous Computing*, 6(5/6):341–357, 2002.
 - [17] J. Bellido, R. Alarcón, and C. Sepulveda. Web Linking-Based Protocols for Guiding RESTful M2M Interaction. In A. Harth and N. Koch, editors, *Current Trends in Web Engineering*, volume 7059 of *Lecture Notes in Computer Science*, pages 74–85. Springer, 2012.
 - [18] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
 - [19] I. Biederman. Recognition-by-Components: A Theory of Human Image Understanding. *Psychological Review*, 94(2):115–147, 1987.
 - [20] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and Magic Lenses: The See-Through Interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 73–80. ACM, 1993.
 - [21] M. Blackstock and R. Lea. IoT mashups with the WoTKit. In *Proceedings of the 3rd IEEE International Conference on the Internet of Things (IOT)*, pages 159–166. IEEE Computer Society, 2012.
 - [22] H. Bohn, A. Bobek, and F. Golatowski. SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different do-

- mains. In *Proceedings of the 5th International Conference on Networking, International Conference on Systems, and International Conference on Mobile Communications and Learning Technologies (ICN/ICONS/MCL)*, 2006.
- [23] M. Botts and A. Robin. OpenGIS Sensor Model Language (SensorML) Implementation Specification. Technical report, Open Geospatial Consortium, 2007.
- [24] G. Bovet and J. Hennebert. Offering Web-of-Things Connectivity to Building Networks. In *Adjunct Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 1555–1564. ACM, 2013.
- [25] G. Broll, S. Siorpaes, E. Rukzio, M. Paolucci, J. Hamard, M. Wagner, and A. Schmidt. Supporting Mobile Service Usage through Physical Mobile Interaction. In *Proceedings of the 5th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 262–271. IEEE Computer Society, 2007.
- [26] J. Brooke. SUS: A quick and dirty usability scale. In P. W. Jordan, B. Thomas, B. A. Weerdmeester, and A. L. McClelland, editors, *Usability Evaluation in Industry*. Taylor and Francis, 1996.
- [27] M. Brown and D. G. Lowe. Automatic Panoramic Image Stitching Using Invariant Features. *International Journal of Computer Vision*, 74(1):59–73, 2007.
- [28] J. S. Bruner and L. Postman. On the Perception of Incongruity: A Paradigm. *Journal of Personality*, 18(2):206–223, 1949.
- [29] A. J. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. Home Automation in the Wild: Challenges and Opportunities. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems (CHI)*, pages 2115–2124. ACM, 2011.
- [30] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. BRIEF: Binary Robust Independent Elementary Features. In *Proceedings of the 11th European Conference on Computer Vision (ECCV)*, pages 778–792, 2010.
- [31] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers*, 15(3):289–308, 2003.
- [32] F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and Dynamic Service Composition in *eFlow*. In J. A. B. Jr., J. Krogstie, O. Pastor, B. Pernici, C. Rolland, and A. Sølvsberg, editors, *Seminal Contributions to Information Systems Engineering*, pages 215–233. Springer, 2013.
- [33] A. Charfi, T. Dinkelaker, and M. Mezini. A Plug-in Architecture for Self-Adaptive Web Service Compositions. In *IEEE International Conference on Web Services (ICWS 2009)*, pages 35–42. IEEE Computer Society, 2009.
- [34] A. Charfi, H. Müller, and M. Mezini. Aspect-Oriented Business Process Modeling with AO4BPMN. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, pages 48–61. 2010.

- [35] H.-S. Choi, J.-Y. Lee, N.-R. Yang, and W.-S. Rhee. User-centric Service Environment for Context Aware Service Mash-up. In *Proceedings of the 2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 388–393. IEEE Computer Society, 2014.
- [36] B. Christophe, V. Verdot, and V. Toubiana. Searching the “Web of Things”. In *Proceedings of the 5th IEEE International Conference on Semantic Computing (ICSC)*, pages 308–315. IEEE Computer Society, 2011.
- [37] M. Colombo, E. D. Nitto, and M. Mauri. SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC)*, pages 191–202, 2006.
- [38] E. Corchado and Á. Herrero. Neural Visualization of Network Traffic Data for Intrusion Detection. *Applied Soft Computing*, 11(2):2042–2056, 2011.
- [39] P. M. Corcoran and J. Desbonnet. Browser-style Interfaces to a Home Automation Network. *IEEE Transactions on Consumer Electronics*, 43(4):1063–1069, 1997.
- [40] I. Corredor, J. F. Martínez, M. S. Familiar, and L. López. Knowledge-Aware and Service-Oriented Middleware for deploying pervasive services. *Journal of Network and Computer Applications*, 35(2):562–576, 2012.
- [41] G. Csurka, C. R. Dance, L. Fan, J. Willamowski, and C. Bray. Visual Categorization with Bags of Keypoints. In *Proceedings of the Workshop on Statistical Learning in Computer Vision (SLCV)*, 2004.
- [42] A. L. Dahl, H. Aanæs, and K. S. Pedersen. Finding the Best Feature Detector-Descriptor Combination. In M. Goesele, Y. Matsushita, R. Sagawa, and R. Yang, editors, *Proceedings of the International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT)*, pages 318–325. IEEE Computer Society, 2011.
- [43] F. D. Davis. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly*, 13:319–340, 1989.
- [44] S. De, B. Christophe, and K. Moessner. Semantic enablers for dynamic digital-physical object associations in a federated node architecture for the Internet of Things. *Ad Hoc Networks*, 18:102–120, 2014.
- [45] P. Debaty, P. Goddi, and A. Vorbau. Integrating the Physical World with the Web to Enable Context-Enhanced Mobile Services. *Mobile Networks and Applications*, 10(4):385–394, 2005.
- [46] K. Derthick, J. Scott, N. Villar, and C. Winkler. Exploring Smartphone-based Web User Interfaces for Appliances. In *Proceedings of the International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI)*, pages 227–236, 2013.
- [47] L. R. Dice. Measures of the Amount of Ecologic Association Between Species. *Ecology*, 26(3):297–302, 1945.

- [48] B. Dober. Exploring Query Augmentation for the SOCRADES Application Service Catalogue. Master's thesis, University of Fribourg, Switzerland, 2009.
- [49] F. Dötzer. Privacy Issues in Vehicular Ad Hoc Networks. In G. Danezis and D. Martin, editors, *Privacy Enhancing Technologies*, volume 3856 of *Lecture Notes in Computer Science*, pages 197–209. Springer, 2006.
- [50] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. pREST: a REST-based Protocol for Pervasive Systems. In *Proceedings of the 2004 IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS)*, pages 340–348. IEEE Computer Society, 2004.
- [51] K. Duan and S. S. Keerthi. Which Is the Best Multiclass SVM Method? An Empirical Study. In *Proceedings of the 6th International Workshop on Multiple Classifier Systems (MCS)*, pages 278–285, 2005.
- [52] A. Dunkels. Full TCP/IP for 8-bit Architectures. In *In Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 85–98. ACM, 2003.
- [53] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. Consistency and scalability in event notification for embedded Web applications. In *Proceedings of the 11th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 89–98. IEEE Computer Society, 2009.
- [54] K. Elgazzar, H. S. Hassanein, and P. Martin. DaaS: Cloud-based Mobile Web Service Discovery. *Pervasive and Mobile Computing*, 13:67–84, 2013.
- [55] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 226–231, 1996.
- [56] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, USA, 2000.
- [57] R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technologies*, 2(2):115–150, 2002.
- [58] E. Fleisch. What is the Internet of Things? An Economic Perspective. Technical Report WP-BIZAPP-053, Auto-ID Labs, 2010.
- [59] D. Flohr and J. Fischer. A Lightweight ID-Based Extension for Marker Tracking Systems. In *Proceedings of the 13th Eurographics Symposium on Virtual Environments (IPT/EGVE)*, pages 59–64, 2007.
- [60] C. Frank, P. Bolliger, F. Mattern, and W. Kellerer. The Sensor Internet at Work: Locating Everyday Items Using Mobile Phones. *Pervasive and Mobile Computing*, 4(3):421–447, 2008.
- [61] S. Frølund, F. Pedone, J. Pruyne, and A. V. Moorsel. Building Dependable Internet Services with E-speak. Technical Report HPL-2000-78, HP Labs, 2000.
- [62] K. Fujii and T. Suda. Semantics-based Context-aware Dynamic Service Composi-

- tion. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2), 2009.
- [63] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock. Automatically Generating Personalized User Interfaces with SUPPLE. *Artificial Intelligence*, 174(12/13):910–950, 2010.
 - [64] F. Gao, M. Zaremba, S. Bhiri, and W. Derguech. Extending BPMN 2.0 with Sensor and Smart Device Business Functions. In *Proceedings of the 20th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 297–302. IEEE Computer Society, 2011.
 - [65] M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format Version 3.0 Reference Manual. Technical Report Logic-92-1, Stanford University, 1992.
 - [66] N. Gershenfeld. *When Things Start to Think*. Henry Holt and Co., Inc., 1999.
 - [67] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals. *Circulation*, 101(23):e215–e220, 2000.
 - [68] A. Gómez-Goiri, P. Orduña, J. Diego, and D. L. de Ipiña. Otsopack: Lightweight semantic framework for interoperable ambient intelligence applications. *Computers in Human Behavior*, 30:460–467, 2014.
 - [69] J. R. Goodall, W. G. Lutters, P. Rheingans, and A. Komlodi. Preserving the Big Picture: Visual Network Traffic Analysis with TNV. In *Proceedings of the IEEE Workshop on Visualization for Computer Security (VizSEC)*, pages 47–54. IEEE Computer Society, 2005.
 - [70] F. Gramegna, S. Ieva, G. Loseto, M. Ruta, F. Scioscia, and E. Di Sciascio. A Lightweight Matchmaking Engine for the Semantic Web of Things. In *Proceedings of the 21st Italian Symposium on Advanced Database Systems (SEBD)*, pages 103–114, 2013.
 - [71] C. Grana, D. Borghesani, M. Manfredi, and R. Cucchiara. A Fast Approach for Integrating ORB Descriptors in the Bag of Words Model. In *Multimedia Content and Mobile Devices*, volume 8776 of *Proceedings of SPIE*, 2013.
 - [72] R. E. Grinter, W. K. Edwards, M. W. Newman, and N. Ducheneaut. The Work to Make a Home Network Work. In *Proceedings of the 9th European Conference on Computer Supported Cooperative Work (ECSCW)*, pages 469–488, 2005.
 - [73] D. Guinard. *A Web of Things Application Architecture – Integrating the Real-World into the Web*. PhD thesis, ETH Zurich, Switzerland, 2011.
 - [74] D. Guinard, M. Fischer, and V. Trifa. Sharing Using Social Networks in a Composable Web of Things. In *Workshop Proceedings of the 8th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 702–707. IEEE Computer Society, 2010.
 - [75] D. Guinard, C. Floerkemeier, and S. Sarma. Cloud Computing, REST and Mashups to Simplify RFID Application Development and Deployment. In *Proceedings of the*

- 2nd International Workshop on the Web of Things (WoT)*. ACM, 2011.
- [76] D. Guinard, I. Ion, and S. Mayer. In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers' Perspective. In *Proceedings of the 8th International Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, pages 326–337, 2011.
- [77] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services. *IEEE Transactions on Services Computing*, 3(3):223–235, 2010.
- [78] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices. In D. Uckelmann, M. Harrison, and F. Michahelles, editors, *Architecting the Internet of Things*, pages 97–129. Springer, 2011.
- [79] D. Guinard, V. Trifa, T. Pham, and O. Liechti. Towards Physical Mashups in the Web of Things. In *Proceedings of the 6th IEEE International Conference on Networked Sensing Systems (INSS 2009)*, pages 196–199. IEEE Computer Society, 2009.
- [80] D. Guinard, M. Weiss, and V. Trifa. Are you Energy-Efficient? Sense it on the Web! In *Adjunct Proceedings of the 7th International Conference on Pervasive Computing (Pervasive)*, 2009.
- [81] S. Han, S. Park, G. M. Lee, and N. Crespi. Extending the Device Profile for Web Services (DPWS) standard using a REST Proxy. *IEEE Internet Computing*, PrePrints, 2014.
- [82] B. Harbelot, H. Arenas, and C. Cruz. Semantics for Spatio-temporal “Smart Queries”. In *Proceedings of the 9th International Conference on Web Information Systems and Technologies (WEBIST 2013)*, pages 316–319. 2013.
- [83] R. Hardy and E. Rukzio. Touch & Interact: Touch-based Interaction of Mobile Phones with Displays. In *Proceedings of the International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI)*, pages 245–254, 2008.
- [84] R. Hardy, E. Rukzio, P. Holleis, and M. Wagner. Mobile Interaction with Static and Dynamic NFC-based Displays. In *Proceedings of the 12th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI)*, pages 123–132, 2010.
- [85] W. Harrison. From the Editor: The Dangers of End-User Programming. *IEEE Software*, 21:5–7, 2004.
- [86] J. Helms, R. Schaefer, K. Luyten, J. Vermeulen, M. Abrams, A. Coyette, and J. Vanderdonckt. Human-Centered Engineering of Interactive Systems with the User Interface Markup Language. In A. Seffah, J. Vanderdonckt, and M. C. Desmarais, editors, *Human-Centered Software Engineering – Software Engineering Models, Patterns and Architectures for HCI*, Human-Computer Interaction Series, pages 139–

171. Springer, 2009.
- [87] V. Heun, S. Kasahara, and P. Maes. Smarter Objects: Using AR technology to Program Physical Objects and their Interactions. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems (CHI Extended Abstracts)*, pages 961–966. ACM, 2013.
 - [88] T. D. Hodes, R. H. Katz, E. Servan-Schreiber, and L. Rowe. Composable Ad-hoc Mobile Services for Universal Interaction. In L. Pap, K. Sohraby, D. B. Johnson, and C. Rose, editors, *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Networking and Computing (MobiCom)*, pages 1–12. ACM, 1997.
 - [89] J. W. Hui and D. E. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proceedings of the 6th ACM conference on Embedded Networked Sensor Systems (SenSys)*, pages 15–28. ACM, 2008.
 - [90] J. Humble, A. Crabtree, T. Hemmings, K.-P. Åkesson, B. Koleva, T. Rodden, and P. Hansson. Playing with the Bits – User-configuration of Ubiquitous Domestic Environments. In A. K. Dey, A. Schmidt, and J. F. McCarthy, editors, *Proceedings of the Fifth International Conference on Ubiquitous Computing (Seattle, WA, USA)*, volume 2864 of *Lecture Notes in Computer Science*, pages 256–263. Springer, 2003.
 - [91] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadis, M. Autili, M. A. Gerosa, and A. B. Hamida. Service-Oriented Middleware for the Future Internet: State of the Art and Research Directions. *Journal of Internet Services and Applications*, 2(1):23–45, 2011.
 - [92] F. Jammes and H. Smit. Service-Oriented Paradigms in Industrial Automation. *IEEE Transactions on Industrial Informatics*, 1(1):62–70, 2005.
 - [93] M. Jung, E. Hajdarevic, W. Kastner, and A. Jara. Short paper: A Scripting-Free Control Logic Editor for the Internet of Things. In *Proceedings of the 2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 193–194, 2014.
 - [94] A. Kameas, I. Mavrommati, and P. Markopoulos. *Computing in Tangible: Using Artifacts as Components of Ambient Intelligence Environments*, pages 121–142. IOS Press, 2004.
 - [95] J. Kang. A Framework for Mobile Object Recognition of Internet of Things Devices and Inference with Contexts. *Journal of Industrial and Intelligent Information*, 2(1):51–55, 2014.
 - [96] A. Kansal, S. Nath, J. Liu, and F. Zhao. SenseWeb: An Infrastructure for Shared Sensing. *Multimedia*, 14(4):8–13, 2007.
 - [97] K. Kaowthumrong, J. Lebsack, and R. Han. Automated Selection of the Active Device in Interactive Multi-Device Smart Spaces. In *Proceedings of the UbiComp 2002 Workshop on Supporting Spontaneous Interaction in Ubiquitous Computing Settings*, 2000.
 - [98] V. Karavirta, A. Korhonen, L. Malmi, and T. L. Naps. A Comprehensive Taxonomy of Algorithm Animation Languages. *Visual Languages and Computing*, 21(1):1–22,

- 2010.
- [99] A. Katasonov, O. Kaykova, O. Khriyenko, S. Nikitin, and V. Y. Terziyan. Smart Semantic Middleware for the Internet of Things. In *Proceedings of the 5th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, pages 169–178, 2008.
 - [100] W. C. Kim and J. D. Foley. Providing High-level Control and Expert Assistance in the User Interface Presentation Design. In S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel, and T. N. White, editors, *Proceedings of the Human-Computer Interaction and Human Factors in Computing Conferences (INTERCHI)*, pages 430–437. ACM, 1993.
 - [101] T. Kindberg and J. Barton. A Web-based Nomadic Computing System. *Computer Networks*, 35(4):443–456, 2001.
 - [102] W. Kleiminger, C. Beckel, and S. Santini. Opportunistic Sensing for Efficient Energy Usage in Private Households. In *Proceedings of the Smart Energy Strategies Conference 2011 (SES)*, 2011.
 - [103] G. Klein and D. W. Murray. Improving the Agility of Keyframe-Based SLAM. In *Proceedings of the 10th European Conference on Computer Vision (ECCV)*, pages 802–815, 2008.
 - [104] M. Klusch and A. Gerber. Semantic Web Service Composition Planning with OWLS-XPlan. In *Proceedings of the 1st International AAI Fall Symposium on Agents and the Semantic Web*, pages 55–62, 2005.
 - [105] J. Kopecký, K. Gomadam, and T. Vitvar. hRESTS: An HTML Microformat for Describing RESTful Web Services. pages 619–625. IEEE Computer Society, 2008.
 - [106] M. Kovatsch. CoAP for the Web of Things: From Tiny Resource-constrained Devices to the Web Browser. In *Adjunct Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 1495–1504. ACM, 2013.
 - [107] M. Kovatsch, S. Mayer, and B. Ostermaier. Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In *Proceedings of the 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, pages 751–756, 2012.
 - [108] M. Kranz, A. Schmidt, and P. Holleis. Embedded Interaction – Interacting with the Internet of Things. *IEEE Internet Computing*, 99(1):46–53, 2009.
 - [109] M. Langheinrich. *Personal Privacy in Ubiquitous Computing – Tools and System Support*. PhD thesis, ETH Zurich, Switzerland, 2005.
 - [110] M. Lanthaler and C. Guetl. Hydra: A Vocabulary for Hypermedia-Driven Web APIs. In C. Bizer, T. Heath, T. Berners-Lee, M. Hausenblas, and S. Auer, editors, *Proceedings of the WWW 2013 Workshop on Linked Data on the Web (LDOW)*, number 996 in CEUR Workshop Proceedings, 2013.
 - [111] J. L. M. Lastra and I. M. Delamer. Semantic Web Services in Factory Automation:

- Fundamental Insights and Research Roadmap. *IEEE Transactions on Industrial Informatics*, 2(1):1–11, 2006.
- [112] F. Lécué, Y. Gorronogoitia, R. Gonzalez, M. Radzinski, and M. Villa. SOA4All: An Innovative Integrated Approach to Services Composition. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 58–67. IEEE Computer Society, 2010.
- [113] S. Leutenegger, M. Chli, and R. Siegwart. BRISK: Binary Robust Invariant Scalable Keypoints. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2548–2555. IEEE Computer Society, 2011.
- [114] J. R. Lewis and J. Sauro. The Factor Structure of the System Usability Scale. In *Proceedings of the 1st International Conference on Human Centered Design (HCD)*, volume 5619 of *Lecture Notes in Computer Science*, pages 94–103. Springer, 2009.
- [115] F.-F. Li and P. Perona. A Bayesian Hierarchical Model for Learning Natural Scene Categories. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 524–531. IEEE Computer Society, 2005.
- [116] A. Lorenz. Architectural patterns for applications with external user interface elements. *Pervasive and Mobile Computing*, 9(2):269–280, 2013.
- [117] T. Lotan and T. Toledo. An In-Vehicle Data Recorder for Evaluation of Driving Behavior and Safety. *Transportation Research Record*, 1953:112–119, 2006.
- [118] D. G. Lowe. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the International Conference on Computer Vision (ICCV)*, pages 1150–1157. IEEE Computer Society, 1999.
- [119] Z. Maamar, Q. Z. Sheng, and B. Benatallah. Towards an Approach for Coordinating Personalized Composite Services in an Environment of Mobile Users. In *Revised Selected Papers of the Ubiquitous Mobile Information and Collaboration Systems (UMICS)*, pages 69–82, 2004.
- [120] L. Mainetti, V. Mighali, L. Patrono, P. Rametta, and S. L. Oliva. A novel architecture enabling the visual implementation of Web of Things applications. In *Proceedings of the 21st International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE Computer Society, 2013.
- [121] V. Majuntke, G. Schiele, K. Spohrer, M. Handte, and C. Becker. A Coordination Framework for Pervasive Applications in Multi-user Environments. In *Proceedings of the 6th International Conference on Intelligent Environments (IE)*, pages 178–184. IEEE Computer Society, 2010.
- [122] C. C. Marshall and F. M. Shipman. Which semantic web? In *Proceedings of the 14th ACM Conference on Hypertext and Hypermedia (HYPERTEXT)*, pages 57–66. ACM, 2003.
- [123] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust Wide Baseline Stereo from Maximally Stable Extremal Regions. In *Proceedings of the British Machine Vision*

- Conference (BMVC)*. British Machine Vision Association, 2002.
- [124] K. Mathieson, E. Peacock, and W. W. Chin. Extending the Technology Acceptance Model: The Influence of Perceived User Resources. *ACM SIGMIS Database*, 32:86–112, 2001.
 - [125] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.
 - [126] F. Mattern and C. Floerkemeier. From the Internet of Computers to the Internet of Things. In K. Sachs, I. Petrov, and P. Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *LNCS*, pages 242–259. Springer, 2010.
 - [127] F. Mattern and P. Sturm. From Distributed Systems to Ubiquitous Computing. In K. Irmscher and K.-P. Fährnich, editors, *Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell, pages 3–25. Springer, 2003.
 - [128] S. Mayer. Deployment Support for an Infrastructure for Web-enabled Devices. Master’s thesis, ETH Zurich, Switzerland, 2010.
 - [129] S. Mayer. Service Integration - A Web of Things Perspective. In *Proceedings of the W3C Workshop on Data and Services Integration*, 2011.
 - [130] S. Mayer. Web-based Service Brokerage for Robotic Devices. In *Adjunct Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp)*, pages 863–865. ACM, 2012.
 - [131] S. Mayer and G. Basler. Semantic Metadata to Support Device Interaction in Smart Environments. In *Adjunct Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 1505–1514. ACM, 2013.
 - [132] S. Mayer and G. Basler. Semantic Metadata to Support Device Interaction in Smart Environments (Poster Abstract). In *Adjunct Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 243–246. ACM, 2013.
 - [133] S. Mayer, C. Beckel, B. Scheidegger, C. Barthels, and G. Sörös. Demo: Uncovering Device Whispers in Smart Homes. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia (MUM)*. ACM, 2012.
 - [134] S. Mayer and D. Guinard. An Extensible Discovery Service for Smart Things. In *Proceedings of the 2nd International Workshop on the Web of Things (WoT)*. ACM, 2011.
 - [135] S. Mayer, D. Guinard, and V. Trifa. Facilitating the Integration and Interaction of Real-World Services for the Web of Things. In *Proceedings of the UrbanIOT 2010 Workshop*, 2010.
 - [136] S. Mayer, D. Guinard, and V. Trifa. Searching in a Web-based Infrastructure for Smart Things. In *Proceedings of the 3rd International Conference on the Internet*

- of Things (IoT)*, pages 119–126. IEEE Computer Society, 2012.
- [137] S. Mayer, Y. N. Hassan, and G. Sörös. A Magic Lens for Revealing Device Interactions in Smart Environments. In *Proceedings of the SIGGRAPH Asia 2014 Symposium on Mobile Graphics and Interactive Applications (MGIA)*, 2014.
 - [138] S. Mayer, N. Inhelder, R. Verborgh, and R. V. de Walle. User-friendly Configuration of Smart Environments. In *Proceedings of the 2014 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 163–165. IEEE Computer Society, 2014.
 - [139] S. Mayer, N. Inhelder, R. Verborgh, R. V. de Walle, and F. Mattern. Configuration of Smart Environments Made Simple – Combining Visual Modeling with Semantic Metadata and Reasoning. In *Proceedings of the 4th International Conference on the Internet of Things (IoT)*, 2014.
 - [140] S. Mayer and D. S. Karam. A Computational Space for the Web of Things. In S. Mayer, D. Guinard, and E. Wilde, editors, *Proceedings of the 3rd International Workshop on the Web of Things (WoT)*. ACM, 2012.
 - [141] S. Mayer, M. Schalch, M. George, and G. Sörös. Device Recognition for Intuitive Interaction with the Web of Things (Poster Abstract). In *Adjunct Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 239–242. ACM, 2013.
 - [142] S. Mayer and G. Sörös. User Interface Beaming – Seamless Interaction with Smart Things using Wearables. In *Proceedings of the Glass and Eyewear Computers Workshop*, 2014.
 - [143] S. Mayer, A. Tschöfen, A. K. Dey, and F. Mattern. User Interfaces for Smart Things – A Generative Approach with Semantic Interaction Descriptions. *ACM Transactions on Computer-Human Interaction*, 21(2), 2014.
 - [144] D. McDermott. Estimated-Regression Planning for Interactions with Web Services. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 204–211. AAAI, 2002.
 - [145] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL – The Planning Domain Definition Language. Technical Report CVC TR98003/DCS TR1165, Yale Center for Computational Vision and Control, 1998.
 - [146] S. A. McIlraith and T. C. Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 482–496, 2002.
 - [147] B. Medjahed and A. Bouguettaya. A Multilevel Composability Model for Semantic Web Services. *IEEE Transactions on Knowledge and Data Engineering*, 17(7):954–968, 2005.
 - [148] N. Mehandjiev, A. Namoun, F. Lécué, U. Wajid, and G. Kleanthous. End Users Developing Mashups. In *Web Services Foundations*, pages 709–736. Springer, 2014.

- [149] G. Meixner, F. Paternò, and J. Vanderdonckt. Past, Present, and Future of Model-Based User Interface Development. *i-com*, 10(3):2–11, 2011.
- [150] A. Messer, A. Kunjithapatham, M. Sheshagiri, H. Song, P. Kumar, P. Nguyen, and K. H. Yi. InterPlay: A Middleware for Seamless Device Integration and Task Orchestration in a Networked Home. In *Proceedings of the 4th IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 296–307. IEEE Computer Society, 2006.
- [151] S. Meyer, A. Ruppen, and C. Magerkurth. Internet of Things-Aware Process Modeling: Integrating IoT Devices as Business Process Resources. In *Proceedings of the 25th International Conference on Advanced Information Systems Engineering (CAiSE)*, Lecture Notes in Computer Science, pages 84–98. Springer, 2013.
- [152] N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6):51–59, 2004.
- [153] P. Minarik and T. Dymacek. NetFlow Data Visualization Based on Graphs. In *Proceedings of the 5th International Workshop on Visualization for Computer Security (VizSec)*, volume 5210 of *Lecture Notes in Computer Science*, pages 144–151. Springer, 2008.
- [154] K. Mitchell, N. J. P. Race, and M. Clarke. CANVIS: Context-Aware Network Visualization Using Smartphones. In *Proceedings of the 7th Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI)*, pages 175–182, 2005.
- [155] S. B. Mokhtar, P.-G. Raverdy, A. Urbietta, and R. S. Cardoso. Interoperable Semantic and Syntactic Service Discovery for Ambient Computing Environments. *International Journal of Ambient Computing and Intelligence*, 2(4):13–32, 2010.
- [156] P. Moreels and P. Perona. Evaluation of Features Detectors and Descriptors Based on 3D Objects. *International Journal of Computer Vision*, 73(3):263–284, 2007.
- [157] O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, pages 815–824. ACM, 2008.
- [158] B. Myers, S. E. Hudson, and R. Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, 7(1):3–28, 2000.
- [159] B. A. Myers. A New Model for Handling Input. *ACM Transactions on Information Systems*, 8(3):289–320, 1990.
- [160] D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni. ICOs: A Model-Based User Interface Description Technique Dedicated to Interactive Systems Addressing Usability, Reliability and Scalability. *ACM Transactions on Computer-Human Interaction*, 16(4), 2009.
- [161] J. Nichols and B. A. Myers. Creating a Lightweight User Interface Description Language: An Overview and Analysis of the Personal Universal Controller Project.

- ACM Transactions on Computer-Human Interaction*, 16(4), 2009.
- [162] J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. Generating Remote Control Interfaces for Complex Appliances. In *Proceedings of the 2002 International Conference on Intelligent User Interfaces (IUI)*, pages 161–170. ACM, 2002.
 - [163] J. Nichols, B. A. Myers, and K. Litwack. Improving Automatic Interface Generation with Smart Templates. In J. Vanderdonckt, N. J. Nunes, and C. Rich, editors, *Proceedings of the 2004 International Conference on Intelligent User Interfaces (IUI)*, pages 286–288. ACM, 2004.
 - [164] D. R. Olsen, S. Jefferies, T. Nielsen, W. Moyes, and P. Fredrickson. Cross-modal Interaction using XWeb. In *Proceedings of the 2000 International Conference on Intelligent User Interfaces (IUI)*, pages 191–200. ACM, 2000.
 - [165] M. O’Neill. The Internet of Things: do more devices mean more risks? *Computer Fraud & Security*, 2014(1):16–17, 2014.
 - [166] B. Ostermaier, M. Kovatsch, and S. Santini. Connecting Things to the Web using Programmable Low-power WiFi Modules. In *Proceedings of the 2nd International Workshop on the Web of Things (WoT)*. ACM, 2011.
 - [167] B. Ostermaier, K. Römer, F. Mattern, M. Fahrmaier, and W. Kellerer. A Real-Time Search Engine for the Web of Things. In *Proceedings of the 2nd IEEE International Conference on the Internet of Things (IoT)*. IEEE Computer Society, 2010.
 - [168] J. Paefgen, T. Staake, and F. Thiesse. Resolving the Misalignment between Consumer Privacy Concerns and Ubiquitous IS Design: The Case of Usage-based Insurance. In *Proceedings of the 33rd International Conference on Information Systems (ICIS)*, 2012.
 - [169] K. Paridel, E. Bainomugisha, Y. Vanrompay, Y. Berbers, and W. D. Meuter. Middleware for the Internet of Things, Design Goals and Challenges. In *Proceedings of the 3rd International DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS)*, 2010.
 - [170] F. Paternò. Model-based Tools for Pervasive Usability. *Interacting with Computers*, 17(3):291–315, 2005.
 - [171] F. Paternò, C. Santoro, and L. D. Spano. MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments. *ACM Transactions on Computer-Human Interaction*, 16(4), 2009.
 - [172] A. Pattath, B. D. Bue, Y. Jang, D. S. Ebert, X. Zhong, A. Ault, and E. J. Coyle. Interactive Visualization and Analysis of Network and Sensor Data on Mobile Devices. In *Proceedings of the IEEE Symposium On Visual Analytics Science And Technology (VAST)*, pages 83–90, 2006.
 - [173] C. Pautasso. Composing RESTful services with JOpera. In A. Bergel and J. Fabry, editors, *Proceedings of the 8th International Conference on Software Composition (SC)*, volume 5634 of *Lecture Notes in Computer Science*, pages 142–159. Springer,

- 2009.
- [174] C. Pautasso and E. Wilde. Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In *Proceedings of the 18th International World Wide Web Conference (WWW)*, pages 911–920. ACM, 2009.
 - [175] C. Pautasso and E. Wilde. Push-Enabling RESTful Business Processes. In *Proceedings of the 9th International Conference on Service-Oriented Computing (ICSOC)*, pages 32–46, 2011.
 - [176] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, pages 805–814. ACM, 2008.
 - [177] C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10):46–52, 2003.
 - [178] C. Perera, A. B. Zaslavsky, P. Christen, M. Compton, and D. Georgakopoulos. Context-Aware Sensor Search, Selection and Ranking Model for Internet of Things Middleware. In *IEEE 14th International Conference on Mobile Data Management (MDM)*, pages 314–322. IEEE Computer Society, 2013.
 - [179] A. Pintus, D. Carboni, and A. Piras. Paraimpu: a Platform for a Social Web of Things. In *Proceedings of the 21st World Wide Web Conference (WWW, Companion Volume)*, pages 401–404. ACM, 2012.
 - [180] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proceedings of the 11th International World Wide Web Conference (WWW)*. ACM, 2002.
 - [181] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In G. D. Abowd, B. Brumitt, and S. A. Shafer, editors, *Proceedings of the 3rd International Conference on Ubiquitous Computing (Ubicomp)*, volume 2201 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2001.
 - [182] E. S. Poole, M. Chetty, R. E. Grinter, and W. K. 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)*, pages 455–464. ACM, 2008.
 - [183] B. A. Price, R. M. Baecker, and I. S. Small. A Principled Taxonomy of Software Visualization. *Visual Languages and Computing*, 4(3):211–266, 1993.
 - [184] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny Web Services: Design and Implementation of Interoperable and Evolvable Sensor Networks. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 253–266. ACM, 2008.
 - [185] D. Randall. Living Inside a Smart Home: A Case Study. In R. Harper, editor, *Inside the Smart Home*, pages 227–246. Springer, 2003.
 - [186] P.-G. Raverdy, O. Riva, A. de La Chapelle, R. Chibout, and V. Issarny. Efficient

- Context-aware Service Discovery in Multi-Protocol Pervasive Environments. In *Proceedings of the 7th International Conference on Mobile Data Management (MDM)*, 2006.
- [187] D. Reilly, M. Welsman-Dinelle, C. Bate, and K. Inkpen. Just Point and Click? Using Handhelds to Interact with Paper Maps. In *Proceedings of the 7th international conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI)*, pages 239–242. ACM, 2005.
- [188] A. Rial and G. Danezis. Privacy-preserving Smart Metering. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 49–60. ACM, 2011.
- [189] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, Inc, First edition, 2007.
- [190] T. Riedel, N. Fantana, A. Genaid, D. Yordanov, H. R. Schmidtke, and M. Beigl. Using Web Service Gateways and Code Generation for Sustainable IoT System Development. In *Proceedings of the Internet of Things 2010 Conference (IoT)*. IEEE Computer Society, 2010.
- [191] M. Rietzler, J. Greim, M. Walch, F. Schaub, B. Wiedersheim, and M. Weber. HomeBLOX: Introducing Process-Driven Home Automation. In *Adjunct Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 801–808. ACM, 2013.
- [192] C. Röcker, M. D. Janse, N. Portolan, and N. Streitz. User Requirements for Intelligent Home Environments: A Scenario-Driven Approach and Empirical Cross-Cultural Study. In *Proceedings of the 2005 Joint Conference on Smart Objects and Ambient Intelligence (sOc-EUSAI)*, pages 111–116. ACM, 2005.
- [193] C. Roduner. *Mobile Devices for Interacting with Tagged Objects: Development Support and Usability*. PhD thesis, ETH Zurich, Switzerland, 2010.
- [194] C. Roduner, M. Langheinrich, C. Floerkemeier, and B. Schwarzentrub. Operating Appliances with Mobile Phones - Strengths and Limits of a Universal Interaction Device. In *Proceedings of the 5th International Conference on Pervasive Computing (Pervasive)*, volume 4480 of *Lecture Notes in Computer Science*, pages 198–215. Springer, 2007.
- [195] S. Rogers, P. Langley, and C. Wilson. Mining GPS Data to Augment Road Models. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 104–113. ACM, 1999.
- [196] R. Roman and J. Lopez. Integrating Wireless Sensor Networks and the Internet: A Security Analysis. *Internet Research: Electronic Networking Applications and Policy*, 19(2):246–259, 2009.
- [197] K. Römer, B. Ostermaier, F. Mattern, M. Fahrmaier, and W. Kellerer. Real-Time Search for Real-World Entities: A Survey. *Proceedings of the IEEE*, 98(11):1887–1902, 2010.

- [198] F. Rosenberg, F. Curbera, M. J. Duftler, and R. Khalaf. Composing RESTful Services and Collaborative Workflows: A Lightweight Approach. *IEEE Internet Computing*, 12(5):24–31, 2008.
- [199] P. L. Rosin. Measuring Corner Properties. *Computer Vision and Image Understanding*, 73(2):291–307, 1999.
- [200] E. Rosten and T. Drummond. Machine Learning for High-Speed Corner Detection. In *Proceedings of the 9th European Conference on Computer Vision (ECCV 2006)*, Lecture Notes in Computer Science, pages 430–443. Springer, 2006.
- [201] E. Rublee, V. Rabaud, K. Konolige, and G. R. Bradski. ORB: An efficient alternative to SIFT or SURF. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2564–2571. IEEE Computer Society, 2011.
- [202] E. Rukzio, K. Leichtenstern, V. Callaghan, P. Holleis, A. Schmidt, and J. Chin. An Experimental Comparison of Physical Mobile Interaction Techniques: Touching, Pointing and Scanning. In *Proceedings of the 8th International Conference on Ubiquitous Computing (Ubicomp)*, volume 4206 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2006.
- [203] G. Schall, S. Zollmann, and G. Reitmayr. Smart Vidente: advances in mobile augmented reality for interactive visualization of underground infrastructure. *Personal and Ubiquitous Computing*, 17(7):1533–1549, 2013.
- [204] R. C. Schank and L. Tesler. A Conceptual Dependency Parser for Natural Language. In *Proceedings of the 1969 Conference on Computational Linguistics (COLING)*. Association for Computational Linguistics, 1969.
- [205] C. Seeger, A. Buchmann, and K. Van Laerhoven. An Event-based BSN Middleware that supports Seamless Switching between Sensor Configurations. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium (IHI)*, pages 503–512. ACM, 2012.
- [206] N. Shadbolt, T. Berners-Lee, and W. Hall. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.
- [207] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu. Web services composition: A decade’s overview. *Information Sciences*, 280:218–238, 2014.
- [208] A. P. Sheth, K. Gomadam, and J. Lathem. SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. *IEEE Internet Computing*, 11:91–94, 2007.
- [209] W. Shi and Y. E. Liu. Real-time urban traffic monitoring with global positioning system-equipped vehicles. *Intelligent Transport Systems*, 4(2):113, 2010.
- [210] K. Sivashanmugam, K. Verma, A. P. Sheth, and J. A. Miller. Adding Semantics to Web Services Standards. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 395–401. CSREA Press, 2003.
- [211] D. Skogan, R. Grønmo, and I. Solheim. Web Service Composition in UML. In *Proceedings of the 8th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 47–57. IEEE Computer Society, 2004.

- [212] R. C. Smith, S. Shahidinejad, D. Blair, and E. L. Bibeau. Characterization of urban commuter driving profiles to optimize battery size in light-duty plug-in electric vehicles. *Transportation Research Part D: Transport and Environment*, 16(3):218–224, 2011.
- [213] S. Sohrabi, N. Prokoshyna, and S. A. McIlraith. Web Service Composition via the Customization of Golog Programs with User Preferences. In *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2009.
- [214] S. Srinivasan, S. Bricka, and C. Bhat. Methodology for Converting GPS Navigational Streams to the Travel-Diary Data Format. Technical report.
- [215] T. Starner. Project Glass: An Extension of the Self. *IEEE Pervasive Computing*, 12(2):14–16, 2013.
- [216] P. Stephan, M. Eich, J. Neidig, M. Rosjat, and R. Hengst. Applying Digital Product Memories in Industrial Production. In W. Wahlster, editor, *SemProM: Foundations of Semantic Product Memories for the Internet of Things*, pages 283–304. Springer, 2013.
- [217] V. Stirbu. Towards a RESTful Plug and Play Experience in the Web of Things. In *Proceedings of the 2nd IEEE International Conference on Semantic Computing (ICSC)*, pages 512–517. IEEE Computer Society, 2008.
- [218] N. A. Streitz. Ubiquitous Computing and The Disappearing Computer – Research Agendas, Issues, and Strategies. In *Proceedings of the 3rd International Conference on Ubiquitous Computing (UbiComp)*, volume 2201 of *Lecture Notes in Computer Science*, pages 184–186. Springer, 2001.
- [219] L. Takayama, C. Pantofaru, D. Robson, B. Soto, and M. Barry. Making Technology Homey: Finding Sources of Satisfaction and Meaning in Home Automation. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp)*, pages 511–520. ACM, 2012.
- [220] C. C. Tan, B. Sheng, H. Wang, and Q. Li. Microsearch: A Search Engine for Embedded Devices Used in Pervasive Computing. *ACM Transactions on Embedded Computing Systems*, 9(4), 2010.
- [221] K. Thoelen, S. Michiels, and W. Joosen. On-Demand Attribute-Based Service Discovery for Mobile WSAWs. In *Proceedings of the 5th International Conference on Communication System Software and Middleware (COMSWARE)*. ACM, 2011.
- [222] E. Tokunaga, H. Kimura, N. Kobayashi, and T. Nakajima. Virtual Tangible Widgets: Seamless Universal Interaction with Personal Sensing Devices. In *Proceedings of the 7th International Conference on Multimodal Interfaces (ICMI 2005)*, pages 325–332. ACM, 2005.
- [223] E. Tola, V. Lepetit, and P. Fua. DAISY: An Efficient Dense Descriptor Applied to Wide Baseline Stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(5):815–830, 2010.

- [224] M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli, and P. Traverso. ASTRO: Supporting Composition and Execution of Web Services. In *Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC)*, volume 3826 of *Lecture Notes in Computer Science*, pages 495–501. Springer, 2005.
- [225] V. Trifa, D. Guinard, P. Bolliger, and S. Wieland. Design of a Web-based Distributed Location-Aware Infrastructure for Mobile Devices. In *Workshop Proceedings of the 8th Annual International Conference on Pervasive Computing and Communications (PerCom)*, pages 714–719. IEEE Computer Society, 2010.
- [226] V. Trifa, D. Guinard, and S. Mayer. Leveraging the Web for a Distributed Location-aware Infrastructure for the Real World. In E. Wilde and C. Pautasso, editors, *REST: From Research to Practice*. Springer, 2011.
- [227] V. Trifa, S. Wieland, D. Guinard, and T. M. Bohnert. Design and Implementation of a Gateway for Web-based Interaction and Management of Embedded Devices. In *Proceedings of the 2nd International Workshop on Sensor Network Engineering (IWSNE)*, 2009.
- [228] G. Vanderhulst, K. Luyten, and K. Coninx. Pervasive Maps: Explore and Interact with Pervasive Environments. In *Proceedings of the 8th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 227–234. IEEE Computer Society, 2010.
- [229] R. Verborgh. *Serendipitous Web Applications through Semantic Hypermedia*. PhD thesis, Ghent University, Belgium, 2014.
- [230] R. Verborgh, V. Haerinck, T. Steiner, D. Van Deursen, S. Van Hoecke, J. De Roo, R. Van de Walle, and J. G. Vallés. Functional Composition of Sensor Web APIs. In *Proceedings of the 5th International Workshop on Semantic Sensor Networks (SSN)*, number 904 in *CEUR Workshop Proceedings*, pages 65–80, 2012.
- [231] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, and R. Van de Walle. Functional Descriptions as the Bridge between Hypermedia APIs and the Semantic Web. In *Proceedings of the 3rd International Workshop on RESTful Design (WS-REST)*, pages 33–40. ACM, 2012.
- [232] R. Verborgh, T. Steiner, D. Van Deursen, R. Van de Walle, and J. G. Vallés. Efficient Runtime Service Discovery and Consumption with Hyperlinked RESTdesc. In *Proceedings of the 7th International Conference on Next Generation Web Services Practices (NWeSP)*, pages 373–379. IEEE Computer Society, 2011.
- [233] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering Similar Multidimensional Trajectories. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 673–684. IEEE Computer Society, 2002.
- [234] R. Want, K. P. Fishkin, A. Gujar, and B. L. Harrison. Bridging Physical and Virtual Worlds with Electronic Tags. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 370–377. ACM, 1999.
- [235] M. Weiser, R. Gold, and J. S. Brown. The origins of ubiquitous computing research

- at PARC in the late 1980s. *IBM Systems*, 38(4):693–696, 1999.
- [236] M. Weiss and D. Guinard. Increasing Energy Awareness Through Web-enabled Power Outlets. In M. C. Angelides, L. Lambrinos, M. Rohs, and E. Rukzio, editors, *Proceedings of the 9th International Conference on Mobile and Ubiquitous Multimedia (MUM)*. ACM, 2010.
 - [237] S. A. White. Using BPMN to Model a BPEL Process. Technical report, IBM Corp., 2009.
 - [238] E. Wilde and M. Kofahl. The Locative Web. In *Proceedings of the 1st International Workshop on Location and the Web (LocWeb)*, pages 1–8. ACM, 2008.
 - [239] E. Wilhelm, J. Siegel, S. Mayer, J. Paefgen, V. Tiefenbeck, M. Bicker, S. Ho, R. Dantu, and S. Sarma. CloudThink: An Open Standard for Projecting Objects into the Cloud. Technical report, Massachusetts Institute of Technology, Eidgenössische Technische Hochschule Zürich, Singapore University of Technology and Design, and the University of North Texas, 2013.
 - [240] D. Yazar and A. Dunkels. Efficient Application Integration in IP-based Sensor Networks. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)*, pages 43–48. ACM, 2009.
 - [241] J. Yu, J. Han, Q. Z. Sheng, and S. O. Gunarso. PerCAS: An Approach to Enabling Dynamic and Personalized Adaptation for Context-Aware Services. In *Proceedings of the 10th International Conference on Service-Oriented Computing (ICSOC)*, volume 7636 of *Lecture Notes in Computer Science*, pages 173–190. Springer, 2012.
 - [242] J. Zantema, D. H. van Amelsfort, M. C. J. Bliemer, and P. H. L. Bovy. Pay-as-You-Drive Strategies: Case Study of Safety and Accessibility Effects. *Transportation Research Record*, 2078:8–16, 2008.
 - [243] F. Zeshan and R. Mohamad. Semantic Web Service Composition: Overview and Limitations. *International Journal on New Computer Architecture and Their Applications*, 1(3):640–651, 2011.
 - [244] K. Zhan, S. Faux, and F. Ramos. Multi-scale Conditional Random Fields for First-Person Activity Recognition. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 51–59. IEEE Computer Society, 2014.
 - [245] H. Zhao and P. Doshi. Towards Automated RESTful Web Service Composition. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 189–196. IEEE Computer Society, 2009.
 - [246] G. Zimmermann, G. Vanderheiden, and A. Gilman. Prototype Implementations for a Universal Remote Console Specification. In *Extended Abstracts of the 2002 Conference on Human Factors in Computing Systems (CHI)*, pages 510–511. ACM, 2002.

REFERENCED WEB RESOURCES

- [247] A9 Inc. OpenSearch 1.1 Specification, Draft 5. Online at http://www.opensearch.org/Specifications/OpenSearch/1.1/Draft_5, 2005.
- [248] Apache Software Foundation. ab – Apache HTTP server benchmarking tool. Online at <http://httpd.apache.org/docs/2.2/programs/ab.html>, 2014.
- [249] ARToolworks, Inc. ARToolKit for Mobile. Online at <http://www.artoolworks.com/products/artoolkit-for-mobile/>, 2014.
- [250] Belkin, Inc. WeMo Home. Online at <http://www.belkin.com/uk/Products/home-automation/c/wemo-home-automation/>, 2014.
- [251] T. Berners-Lee and D. Connolly. Notation3 (N3): A readable RDF syntax. Online at <http://www.w3.org/TeamSubmission/n3/>, 2011.
- [252] Blippar, Inc. Augmented Reality Advertising – Blippar. Online at <https://blippar.com/en/>, 2014.
- [253] B. Boom and M. Herrmann. Introduction to Vision and Robotics Lecture (The University of Edinburgh). Online at <http://www.inf.ed.ac.uk/teaching/courses/ivr/>, 2014.
- [254] R. Brodt. BPEL Designer Project. Online at <http://www.eclipse.org/bpel/>, 2012.
- [255] S. Bushell, P. Bierman, and S. Cheshire. EtherPEG. Online at <http://www.etherpeg.org/>, 2000.
- [256] D. Cederholm and T. Çelik. Microformats Launch Definition. Online at <http://microformats.org/wiki/what-are-microformats>, 2005.
- [257] CoRE Working Group. Constrained Application Protocol (CoAP). Online at <http://tools.ietf.org/html/draft-ietf-core-coap-18>, 2013.
- [258] L. Cruz. After Fukushima: Crowd-Sourcing Initiative Sets Radiation Data Free. Online at http://newsroom.cisco.com/feature/1360403/After-Fukushima-Crowd-Sourcing-Initiative-Sets-Radiation-Data-Free?utm_medium=rss, 2014.

- [259] DCMI Usage Board. Dublin Core Metadata Initiative Metadata Terms. Online at <http://dublincore.org/documents/dcmi-terms/>, 2012.
- [260] Digital Living Network Alliance. DLNA for Industry. Online at <http://www.dlna.org/>, 2014.
- [261] A. DuVander. Say “Bye XML”. Online at <http://www.programmableweb.com/news/1-5-apis-say-bye-xml/2011/05/25>, 2011.
- [262] Ecma International. The JSON Data Interchange Format. Online at <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>, 2013.
- [263] J. Edwards. Apple Is Launching A Vast Project To Map The Inside Of Every Large Building It Can. Online at <http://www.businessinsider.com/apple-indoor-mapping-project-and-ibeacon-2014-6>, 2014.
- [264] European Commission. ASPIRE Project: Advanced Sensors and Lightweight Programmable Middleware for Innovative RFID Enterprise Applications. Online at <http://www.fp7-aspire.eu/>, 2007.
- [265] European Commission. SMART Project: Search Engine for Multimedia Environment Generated Content. Online at <http://www.smartfp7.eu>, 2007.
- [266] European Commission. Internet of Things - An action plan for Europe. Online at <http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2009:0278:FIN:EN:PDF>, 2009.
- [267] European Commission. SOCRADES Project: Service-Oriented Cross-Layer Infrastructure for Distributed Smart Embedded Devices. Online at <http://www.socrades.eu/>, 2009.
- [268] European Commission. Internet of Things Architecture. Online at <http://www.iot-a.eu/public>, 2010.
- [269] European Commission. AESOP Project: Architecture for Service-Oriented Process. Online at <http://www.imc-aesop.eu/>, 2011.
- [270] D. Evans. The Internet of Things. Online at <http://blogs.cisco.com/news/the-internet-of-things-infographic/>, 2011.
- [271] Evrythng, Inc. EVERYTHING. Online at <https://www.evrythng.com/>, 2014.
- [272] R. T. Fielding. REST APIs must be hypertext-driven. Online at <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>, 2008.
- [273] J. Formo. A Social Web of Things. Online at <http://www.ericsson.com/uxblog/2012/04/a-social-web-of-things/>, 2012.
- [274] V. Fox. Schema.org: Google, Bing & Yahoo unite to make Search Listings Richer through Structured Data. Online at <http://searchengineland.com/schema-org-google-bing-yahoo-unite-79554>, 2011.
- [275] Gartner Inc. Gartner Says It’s the Beginning of a New Era: The Digital Industrial Economy. Online at <http://www.gartner.com/newsroom/id/2602817>, 2013.
- [276] GNOME Project. GLib. Online at <https://developer.gnome.org/glib/stable/>,

- 2014.
- [277] Google Inc. Google Prediction API. Online at <https://developers.google.com/prediction/>, 2014.
 - [278] Google Inc. Google Trends. Online at <http://www.google.com/trends/>, 2014.
 - [279] J. Gregorio. Do we need WADL? Online at <http://bitworking.org/news/193/Do-we-need-WADL>, 2007.
 - [280] T. D. Hanson. uthash – a hash table for C structures. Online at <http://troydhanson.github.io/uthash/>, 2014.
 - [281] <http://wiki.dbpedia.org> Contributors. DBPedia. Online at <http://wiki.dbpedia.org/>, 2014.
 - [282] IBM Corp. A Smarter Planet. Online at <http://www.ibm.com/smarterplanet>, 2009.
 - [283] IETF Network Working Group. Universal Resource Identifiers in WWW. Online at <http://tools.ietf.org/html/rfc1630>, 1994.
 - [284] IETF Network Working Group. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. Online at <http://tools.ietf.org/html/rfc2045>, 1996.
 - [285] IETF Network Working Group. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. Online at <http://tools.ietf.org/html/rfc2046>, 1996.
 - [286] IETF Network Working Group. Uniform Resource Identifiers (URI): Generic Syntax. Online at <http://tools.ietf.org/html/rfc2396>, 1998.
 - [287] IETF Network Working Group. Uniform Resource Identifier (URI): Generic Syntax. Online at <http://tools.ietf.org/html/rfc3986>, 2005.
 - [288] IFTTT, Inc. IFTTT: Put the Internet to work for you. Online at <https://ifttt.com/>, 2014.
 - [289] International Organization for Standardization. ISO/IEC 29341-1:2011 (UPnP Device Architecture). Online at http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=57195, 2011.
 - [290] Internet Engineering Task Force. vCard MIME Directory Profile. Online at <http://tools.ietf.org/html/rfc2426>, 1998.
 - [291] Internet Engineering Task Force. A JSON Media Type for Describing the Structure and Meaning of JSON Documents. Online at <http://tools.ietf.org/html/draft-zyp-json-schema-03>, 2010.
 - [292] Internet Engineering Task Force. Web Linking. Online at <http://tools.ietf.org/html/rfc5988>, 2010.
 - [293] Internet Engineering Task Force. The OAuth 2.0 Authorization Framework. Online at <http://tools.ietf.org/html/rfc6749>, 2012.
 - [294] Internet Engineering Task Force. CoRE Resource Directory (Internet-Draft). Online at <http://tools.ietf.org/html/draft-ietf-core-resource-directory-01>,

- 2013.
- [295] Internet Engineering Task Force. Media Type Specifications and Registration Procedures. Online at <http://tools.ietf.org/html/rfc6838>, 2013.
 - [296] ioBridge, Inc. ThingSpeak. Online at <https://thingspeak.com/>, 2014.
 - [297] S. Ishino. Wind from Fukushima I. Online at https://play.google.com/store/apps/details?id=jp.gr.java_conf.seigo.stop_ra, 2013.
 - [298] ITEA Cluster Program of the EUREKA Network. SIRENA Project: Service Infrastructure for Real-time Embedded Networked Applications. Online at <https://itea3.org/project/sirena.html>, 2005.
 - [299] iTraff Technology Sp. z o.o. Merchandising through image recognition for developers. Online at <http://blog.recognize.im/index.php/2013/11/merchandising-through-image-recognition-for-developers/#more-751>, 2014.
 - [300] S. Lazebnik. Computer Vision Lecture (The University of North Caroline at Chapel Hill. Online at <http://www.cs.unc.edu/~lazebnik/spring10/>, 2010.
 - [301] LogMeIn, Inc. Xively. Online at <https://xively.com/>, 2014.
 - [302] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic Markup for Web Services (W3C Member Submission). Online at <http://www.w3.org/Submission/OWL-S/>, 2004.
 - [303] B. McClendon. A new frontier for Google Maps: mapping the indoors. Online at <http://googleblog.blogspot.ch/2011/11/new-frontier-for-google-maps-mapping.html>, 2011.
 - [304] Microformats Contributors. Microformats. Online at <http://microformats.org/>, 2014.
 - [305] L. Naef. ClickScript: Easy to Use Visual Programming Language. Online at <http://clickscript.ch>, 2011.
 - [306] Neil Mitchell. HoogLe. Online at www.haskell.org/hoogle, 2013.
 - [307] Nest Labs. Nest. Online at <https://nest.com>, 2014.
 - [308] netfilter.org Contributors. netfilter. Online at <http://www.netfilter.org/>, 2014.
 - [309] Nimbits, Inc. Nimbits. Online at <http://www.nimbits.com>, 2013.
 - [310] Ninja Blocks, Inc. Ninja Sphere. Online at <http://ninjablocks.com>, 2014.
 - [311] OASIS Technical Committees on SCA. Service Component Architecture. Online at <http://www.oasis-open.org/sca>, 2007.
 - [312] OASIS Web Service Discovery and Web Services Devices Profile Technical Committee. Devices Profile for Web Services Version 1.1. Online at <http://docs.oasis-open.org/ws-dd/dpws/1.1/cs-01/wsdd-dpws-1.1-spec-cs-01.html>, 2009.
 - [313] OASIS Web Services Business Process Execution Language Technical Committee.

- Web Services Business Process Execution Language Version 2.0. Online at <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, 2007.
- [314] Open Geospatial Consortium. Geosemantics Domain Working Group. Online at <http://www.opengeospatial.org/projects/groups/semantics>, 2014.
- [315] Open Geospatial Consortium, Geosemantics Domain Working Group. Expanding GeoWeb to an Internet of Things. Online at <http://www.ogcnetwork.net/COMGeoWorkshop>, 2014.
- [316] Open Geospatial Consortium, Geosemantics Domain Working Group. OpenWrt: Linux distribution for embedded devices. Online at <https://openwrt.org/>, 2014.
- [317] OpenCV Contributors. OpenCV: Open Source Computer Vision. Online at <http://opencv.org/>, 2014.
- [318] programmableweb.com Contributors. ProgrammableWeb – APIs, Mashups and the Web as Platform. Online at <http://www.programmableweb.com/>, 2014.
- [319] Project Grizzly Contributors. Project Grizzly. Online at <https://grizzly.java.net/>, 2014.
- [320] Qualcomm, Inc. Augmented Reality Technology and Apps – Vuforia – Qualcomm. Online at qualcomm.com/solutions/augmented-reality, 2013.
- [321] Rick Walter. Fuel Economy Data. Online at <http://www.cert.ucr.edu/events/pems2014/liveagenda/24walter.pdf>, 2014.
- [322] roarmot.co.nz Contributors. ARToolKit Marker Maker. Online at <http://roarmot.co.nz/ar>, 2006.
- [323] J. D. Roo. Euler Yet another proof Engine. Online at <http://eulersharp.sourceforge.net/>, 2014.
- [324] J. Sauro. Measuring Usability with the System Usability Scale. Online at <http://www.measuringusability.com/sus.php>, 2011.
- [325] M. Veloso. PDDL by Example. Online at http://www.cs.toronto.edu/~sheila/384/w11/Assignments/A3/veloso-PDDL_by_Example.pdf, 2002.
- [326] World Wide Web Consortium. Semantic Annotations for WSDL and XML Schema. Online at <http://www.w3.org/TR/sawSDL/>, 2007.
- [327] World Wide Web Consortium. XForms 1.1 . Online at <http://www.w3.org/TR/xforms/>, 2009.
- [328] World Wide Web Consortium. HTTP Vocabulary in RDF 1.0. Online at <http://www.w3.org/TR/HTTP-in-RDF10/>, 2011.
- [329] World Wide Web Consortium. The WebSocket API. Online at <http://dev.w3.org/html5/websockets/>, 2014.
- [330] World Wide Web Consortium and Web Hypertext Application Technology Working Group. HTML5 – A vocabulary and associated APIs for HTML and XHTML. Online at <http://www.w3.org/TR/html5/>, 2014.
- [331] World Wide Web Consortium, HTML Working Group. HTML Microdata. Online

- at <http://www.w3.org/TR/microdata/>, 2013.
- [332] World Wide Web Consortium, Model-Based User Interfaces Working Group. MBUI – Abstract User Interface Models. Online at <http://www.w3.org/TR/abstract-ui/>, 2014.
 - [333] World Wide Web Consortium, OWL Working Group. OWL 2 Web Ontology Language. Online at <http://www.w3.org/TR/owl2-overview/>, 2012.
 - [334] World Wide Web Consortium, RDF Working Group. RDF 1.1 Concepts and Abstract Syntax. Online at <http://www.w3.org/TR/rdf11-concepts/>, 2014.
 - [335] Yahoo! Inc. Pipes: Rewire the Web. Online at <http://pipes.yahoo.com/pipes/>, 2014.
 - [336] Yahoo, Inc. Yahoo Weather – Weather Forecasts. Online at <https://weather.yahoo.com/>, 2014.
 - [337] H. Zhang. Japan Geigermap: At-a-glance. Online at <http://japan.failedrobot.com/>, 2013.

APPENDIX A

Interaction Abstraction Schemas

This appendix contains JSON Schema specifications for the semantic interaction descriptions discussed in Chapter 3 and practical usage examples. Section A lists the abstractions used for describing sensors and Sections A and A those pertaining to stateless and stateful actuators, respectively. The specifications frequently reference helper schemas that are defined in Section A.

Sensor Interaction Abstractions

```
1 {
2   "id": "ch.ethz.inf.vs.wot.ui.getdata",
3   "type": "object",
4   "$schema": "http://json-schema.org/draft-03/schema",
5   "description": "Schema for the get data interaction abstraction",
6   "properties": {
7     "abstraction": {
8       "type": "object",
9       "required": true,
10      "properties": {
11        "$ref": "ch.ethz.inf.vs.wot.ui.abstraction.name"
12      }
13    },
14    "$ref": "ch.ethz.inf.vs.wot.ui.type"
15  }
16 }
```

Listing A.1: JSON schema for the get data interaction abstraction.

```
1 {
2   "type": {
3     "name": "string"
4   },
5   "abstraction": {
6     "name": "get_data"
7   }
8 }
```

Listing A.2: Example for a get data interaction description: display the currently playing song of a hi-fi unit.

```
1 {
2   "id": "ch.ethz.inf.vs.wot.ui.value",
3   "type": "object",
4   "$schema": "http://json-schema.org/draft-03/schema",
5   "description": "Schema for the get value interaction abstraction",
6   "properties": {
7     "abstraction": {
8       "type": "object",
9       "required": true,
10      "properties": {
11        "$ref": "ch.ethz.inf.vs.wot.ui.abstraction.name"
12        "$ref": "ch.ethz.inf.vs.wot.ui.abstraction.anchors"
13      }
14    },
15    "$ref": "ch.ethz.inf.vs.wot.ui.type"
16  }
17 }
```

Listing A.3: Description schema for the get value interaction abstraction.


```
1 {
2   "type":{
3     "name":"number",
4     "range":[ 5, 130 ],
5     "unit":"degrees celsius"
6   },
7   "abstraction":{
8     "name":"get_value",
9     "anchors":[{
10      "name":"positive",
11      "range":[ 20, 130 ]
12    }]
13  }
14 }
```

Listing A.4: Example for a get value interaction description: display the coolant temperature of a car.

```
1 {
2   "id":"ch.ethz.inf.vs.ui.wot.proportion",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the get proportion interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name"
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors"
13      }
14    },
15    "$ref":"ch.ethz.inf.vs.wot.ui.type"
16  }
17 }
```

Listing A.5: Description schema for the get proportion interaction abstraction

```
1 {
2   "type":{
3     "name":"number",
4     "range":[ 0, 100 ],
5     "unit":"%"
6   },
7   "abstraction":{
8     "name":"get_proportion",
9     "anchors":[{
10      "name":"alert",
11      "range":[ 90, 100 ]
12    }]
13  }
14 }
```

Listing A.6: Example for a get_proportion interaction description: monitor the load of a server.

Stateless Actuator Interaction Abstractions

```
1 {
2   "id":"ch.ethz.inf.vs.ui.wot.trigger",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the trigger interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name"
12      }
13    },
14    "$ref":"ch.ethz.inf.vs.wot.ui.type"
15  }
16 }
```

Listing A.7: Description schema for the trigger interaction abstraction.

```
1 {
2   "type":{
3     "name":"string",
4     "label":"Irrigation time"
5   },
6   "abstraction":{
7     "name":"trigger"
8   }
9 }
```

Listing A.8: Example for a trigger interaction description: trigger an irrigation system to switch on for a given time.

```
1 {
2   "id":"ch.ethz.inf.vs.ui.wot.goto",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the goto interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name",
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.orientation",
13        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.rate",
14        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.increment"
15      }
16    },
17    "$ref":"ch.ethz.inf.vs.wot.ui.type"
18  }
19 }
```

Listing A.9: Description schema for the goto interaction abstraction.

```
1 {
2   "type":{
3     "name":"enum",
4     "values":[ "previous", "next" ]
5   },
6   "abstraction":{
7     "name":"goto",
8     "increment":"next",
9     "rate":4,
10    "orientation":"horizontal"
11  }
12 }
```

Listing A.10: Example for a goto interaction description: switch to next or previous song on a hi-fi unit.

Stateful Actuator Interaction Abstractions

```
1 {
2   "id":"ch.ethz.inf.vs.ui.wot.set",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the set interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name",
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors"
13      }
14    },
15    "$ref":"ch.ethz.inf.vs.wot.ui.type"
16  }
17 }
```

Listing A.11: Description schema for the set interaction abstraction.

```
1 {
2   "type":{
3     "name":"string"
4   },
5   "abstraction":{
6     "name":"set"
7   }
8 }
```

Listing A.12: Example for a set interaction description: modify a string in a field.

```
1 {
2   "id":"ch.ethz.inf.vs.wot.ui.setvalue",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the set value interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name",
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors",
13        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.orientation"
14      }
15    },
16    "$ref":"ch.ethz.inf.vs.wot.ui.type"
17  }
18 }
```

Listing A.13: Description schema for the set value interaction abstraction.

```

1 {
2   "type":{
3     "name":"number",
4     "range":[ -20, 50 ],
5     "unit":"degrees celsius"
6   },
7   "abstraction":{
8     "name":"set_value",
9     "anchors":[{
10      "name":"hot",
11      "range":[ 30, 50 ]
12    },
13    {
14      "name":"cold",
15      "range":[ -20, 15 ]
16    }
17  ]
18 }

```

Listing A.14: Example for a set value interaction description: control the setpoint of a smart thermostat.

```

1 {
2   "id":"ch.ethz.inf.vs.wot.ui.level",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the level interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name",
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors",
13        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.neutral",
14        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.orientation"
15      }
16    },
17    "$ref":"ch.ethz.inf.vs.wot.ui.type"
18  }
19 }

```

Listing A.15: Description schema for the level interaction abstraction.

```
1 {
2   "type":{
3     "name":"enum",
4     "values":["darker","neutral","brighter"]
5   },
6   "abstraction" : {
7     "name":"level",
8     "neutral":"neutral"
9   }
10 }
```

Listing A.16: Example for a level interaction description: dim the lights.

```
1 {
2   "id":"ch.ethz.inf.vs.wot.ui.setintensity",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the set intensity interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name",
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors"
13      }
14    },
15    "$ref":"ch.ethz.inf.vs.wot.ui.type"
16  }
17 }
```

Listing A.17: Description schema for the set intensity interaction abstraction.

```
1 {
2   "type":{
3     "name":"integer",
4     "range":[0,100],
5     "unit":"%"
6   },
7   "abstraction":{
8     "name":"set_intensity",
9     "anchors":[{
10      "name":"alert",
11      "range":[ 90, 100]
12    }]
13  }
14 }
```

Listing A.18: Example for a set intensity interaction description: control the volume of a hi-fi unit.

```
1 {
2   "id":"ch.ethz.inf.vs.wot.ui.switch",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the switch interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name",
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors"
13      }
14    },
15    "$ref":"ch.ethz.inf.vs.wot.ui.type"
16  }
17 }
```

Listing A.19: Description schema for the switch interaction abstraction.


```
1 {
2   "type":{
3     "name":"boolean",
4     "values":[ "lock", "unlock" ]
5   },
6   "abstraction":{
7     "name":"switch"
8   }
9 }
```

Listing A.20: Example for a switch interaction description: lock/unlock a car.

```
1 {
2   "id":"ch.ethz.inf.vs.wot.ui.switchmode",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the switch mode interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name",
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors",
13        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.off",
14        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.default"
15      }
16    },
17    "$ref":"ch.ethz.inf.vs.wot.ui.type"
18  }
19 }
```

Listing A.21: Description schema for the switch mode interaction abstraction.

```
1 {
2   "type":{
3     "name":"enum",
4     "values": [ "off", "automatic", "manual" ]
5   },
6   "abstraction":{
7     "name":"switch_mode",
8     "off":"off",
9     "default":"automatic"
10  }
11 }
```

Listing A.22: Example for a switch mode interaction description: control the operating mode of an HVAC system.

```
1 {
2   "id":"ch.ethz.inf.vs.wot.ui.position",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the position interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name",
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors",
13        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.orientation"
14      }
15    },
16    "$ref":"ch.ethz.inf.vs.wot.ui.type"
17  }
18 }
```

Listing A.23: Description schema for the position interaction abstraction.

```
1 {
2   "type":{
3     "name":"integer",
4     "unit":"percent"
5   },
6   "abstraction":{
7     "name":"position",
8     "orientation":"vertical"
9   }
10 }
```

Listing A.24: Example for a position interaction description: control the position of the scrollbar of a Web browser.

```
1 {
2   "id":"ch.ethz.inf.vs.wot.ui.move",
3   "type":"object",
4   "$schema":"http://json-schema.org/draft-03/schema",
5   "description":"Schema for the move interaction abstraction",
6   "properties":{
7     "abstraction": {
8       "type":"object",
9       "required":true,
10      "properties":{
11        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.name",
12        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.anchors",
13        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.neutral",
14        "$ref":"ch.ethz.inf.vs.wot.ui.abstraction.orientation"
15      }
16    },
17    "$ref":"ch.ethz.inf.vs.wot.ui.type"
18  }
19 }
```

Listing A.25: Description schema for the move interaction abstraction.

```
1 {
2   "type":{
3     "name":"enum",
4     "values": [ "down", "stop", "up" ]
5   },
6   "abstraction":{
7     "name":"move",
8     "orientation":"vertical"
9   }
10 }
```

Listing A.26: Example for a move interaction description: control a window blind motor.

Helper Schemas

```
1 {
2   "id":"ch.ethz.inf.vs.wot.ui.abstraction.name",
3   "$schema":"http://json-schema.org/draft-03/schema",
4   "type":"object",
5   "properties":{
6     "name": {
7       "type":"string",
8       "enum":[ "get_data", "get_value", "get_proportion", "trigger", "goto
9         ", "set", "set_value", "level", "set_intensity", "switch_mode", "switch
10        ", "position", "move"]
11     }
12   }
```

Listing A.27: JSON Schema for abstraction name elements.

```
1 {
2   "id": "ch.ethz.inf.vs.wot.ui.type",
3   "name": "type"
4   "$schema": "http://json-schema.org/draft-03/schema",
5   "type": "object",
6   "required": true,
7   "properties": {
8     "name": {
9       "type": "string",
10      "enum": [ "boolean", "enum", "number", "item", "string" ],
11      "required": true
12    },
13    "unit": {
14      "type": "string",
15      "required": false
16    },
17    "label": {
18      "type": "string",
19      "required": false
20    },
21    "$ref": "ch.ethz.inf.vs.wot.ui.range",
22    "values": {
23      "type": "array",
24      "required": false,
25      "items": {
26        "type": "string"
27      }
28    }
29  }
30 }
```

Listing A.28: JSON Schema for type elements.

```
1 {
2   "id": "ch.ethz.inf.vs.wot.ui.abstraction.orientation",
3   "$schema": "http://json-schema.org/draft-03/schema",
4   "type": "object",
5   "properties": {
6     "type": "string",
7     "enum": [ "horizontal", "vertical" ]
8   }
9 }
```

Listing A.29: JSON Schema for orientation elements.

```
1 {
2   "name": "anchors",
3   "id": "ch.ethz.inf.vs.wot.ui.anchors",
4   "required": false,
5   "type": "array",
6   "items": {
7     "id": "ch.ethz.inf.vs.wot.ui.anchor",
8     "required": false,
9     "type": "object",
10    "properties": {
11      "name": {
12        // The permissible values for this attribute
13        // can be adapted to the application at hand
14        "type": "string",
15        "enum": [ "alert", "positive", "negative" ]
16      },
17      "range": {
18        "type": "array",
19        "required": false,
20        "items": {
21          "type": "number"
22        },
23        "minItems": 2,
24        "maxItems": 2
25      },
26      "value": {
27        "type": "string",
28        "required": false
29      }
30    }
31  }
32 }
```

Listing A.30: JSON Schema for anchor elements.

```
1 {
2   "name": "range",
3   "id": "ch.ethz.inf.vs.wot.ui.range",
4   "$schema": "http://json-schema.org/draft-03/schema",
5   "type": "array",
6   "required": false,
7   "items": {
8     "type": "number"
9   },
10  "minItems": 2,
11  "maxItems": 2
12 }
```

Listing A.31: JSON Schema for range elements.

```
1 {
2   // These three items have the same semantics
3   "name": "{off | neutral | default}",
4   "id": "ch.ethz.inf.vs.wot.ui.abstraction.{off | neutral | default}",
5   "$schema": "http://json-schema.org/draft-03/schema",
6   "type": [ "string", "integer", "number" ],
7   "required": false
8 }
```

Listing A.32: JSON Schema for neutral, off, and default elements.

```
1 {
2   "name": "unit",
3   "id": "ch.ethz.inf.vs.wot.ui.unit",
4   "$schema": "http://json-schema.org/draft-03/schema",
5   "type": "string",
6   "required": false
7 }
```

Listing A.33: JSON Schema for unit elements.

```
1 {
2   "name": "rate",
3   "id": "ch.ethz.inf.vs.wot.ui.abstraction.rate",
4   "$schema": "http://json-schema.org/draft-03/schema",
5   "type": [ "integer", "number" ],
6   "required": false
7 }
```

Listing A.34: JSON Schema for rate elements.

APPENDIX B

Evaluation Results: Visual Object Recognition

This appendix contains the detailed results of our application of the ORB and FREAK (FAST) feature detectors/descriptors to images of the objects in our test set, as described in Chapter 4.

ORB detector/descriptor (Table B.1):

- 16 objects
- 20 training images per object
- 190 test images per object
- 200 feature points per image
- Pyramid scale factor 1.2, 4 pyramid levels
- Bag of Words codebook size 1600

FREAK descriptor with FAST detector (Table B.2):

- 16 objects
- 20 training images per object
- 190 test images per object
- FREAK pattern scale factor 22
- 4 octaves
- Bag of Words codebook size 1600

Predicted class																	
	Drone	Keyboard	LegoSpaceship	SemWebBook	Mug	SwissKnife	ComputerMouse	PhoneOffice	SensorBoard	SmartThermostat	MicroClient	WaterBottle	Kettle	LegoRobot	LoudspeakerOffice	SunSPOT	Precision
Drone	175	0	13	0	1	0	0	0	0	1	0	0	0	0	0	0	92.1053%
Keyboard	0	47	0	58	0	0	0	51	0	0	0	0	0	0	34	0	24.7368%
LegoSpaceship	28	0	151	0	1	0	0	0	0	0	0	4	6	0	0	0	79.4737%
SemWebBook	0	0	0	190	0	0	0	0	0	0	0	0	0	0	0	0	100%
Mug	21	0	7	0	152	0	0	0	0	0	3	0	7	0	0	0	80%
SwissKnife	0	0	0	0	0	188	0	0	0	0	2	0	0	0	0	0	98.9474%
ComputerMouse	0	0	0	4	0	1	60	0	10	0	10	0	0	0	105	0	31.5789%
PhoneOffice	0	0	0	0	0	0	0	190	0	0	0	0	0	0	0	0	100%
SensorBoard	0	0	0	1	2	3	0	0	153	6	9	0	3	0	0	13	80.5263%
SmartThermostat	0	0	0	0	0	0	0	0	0	190	0	0	0	0	0	0	100%
MicroClient	0	0	0	0	0	0	0	0	0	0	190	0	0	0	0	0	100%
WaterBottle	3	0	11	22	10	0	59	0	12	29	0	14	27	1	0	2	7.36842%
Kettle	1	5	5	1	10	4	18	0	5	2	21	0	100	0	18	0	52.6316%
LegoRobot	14	0	48	0	4	0	0	0	0	7	0	91	12	14	0	0	7.36842%
LoudspeakerOffice	1	1	4	0	25	25	0	33	0	10	6	23	3	0	32	27	16.8421%
SunSPOT	6	0	1	0	0	7	19	3	0	0	0	0	0	0	5	149	78.4211%

Table B.1: Confusion matrix for visual object recognition using the ORB feature detector/descriptor.

Actual class	Predicted class																Precision
	Drone	Keyboard	LegoSpaceship	SemWebBook	Mug	SwissKnife	ComputerMouse	PhoneOffice	SensorBoard	SmartThermostat	MicroClient	WaterBottle	Kettle	LegoRobot	LoudspeakerOffice	SunSPOT	
Drone	190	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100%
Keyboard	0	190	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100%
LegoSpaceship	52	0	132	0	0	0	0	0	0	0	0	0	0	6	0	0	69.4737%
SemWebBook	0	0	0	190	0	0	0	0	0	0	0	0	0	0	0	0	100%
Mug	12	1	0	2	110	0	0	0	0	2	4	2	47	10	0	0	57.8947%
SwissKnife	0	17	0	8	0	158	2	0	2	0	2	0	0	1	0	0	83.1579%
ComputerMouse	0	0	0	0	0	13	127	5	9	0	21	0	0	0	14	1	66.8421%
PhoneOffice	9	0	0	0	0	0	0	181	0	0	0	0	0	0	0	0	95.2632%
SensorBoard	43	0	0	0	0	2	1	0	115	0	0	0	0	29	0	0	60.5263%
SmartThermostat	0	0	0	0	0	0	0	0	0	190	0	0	0	0	0	0	100%
MicroClient	0	0	0	0	0	2	0	0	0	0	188	0	0	0	0	0	98.9474%
WaterBottle	16	53	19	0	11	1	6	0	0	6	0	71	5	2	0	0	37.3684%
Kettle	4	0	0	5	3	10	67	0	0	0	70	0	24	0	7	0	12.6316%
LegoRobot	13	0	47	0	0	0	0	0	0	2	0	77	0	51	0	0	26.8421%
LoudspeakerOffice	3	0	1	22	0	4	28	28	0	1	32	51	9	0	3	8	1.57895%
SunSPOT	177	0	4	0	0	0	0	0	0	0	0	0	0	1	0	8	4.21053%

Table B.2: Confusion matrix for visual object recognition using the FREAK feature descriptor with a FAST feature detector.

APPENDIX C

Free and Open-source Software

Large parts of the work presented in this thesis have been established with the help of free and open-source software and we take this opportunity to thank the contributors of each project that we used when creating it (in alphabetical order):

- Bash (<http://www.gnu.org/s/bash/>), a Unix shell.
- Eclipse (<https://www.eclipse.org/>), an IDE, and its plug-ins.
- Evince (<https://gnome.org/projects/evince>), a document viewer.
- Firefox (<https://www.mozilla.org/firefox/>), a Web browser.
- gedit (<https://gnome.org/projects/gedit/>), a text editor.
- GIMP (<http://www.gimp.org/>), an image manipulation program.
- git (<http://git-scm.com/>), a distributed revision control and source code management system.
- GNU Compiler Collection (<https://gcc.gnu.org/>), a compiler system.
- Gnuplot (<http://www.gnuplot.info/>), a data plotting program.
- Apache HTTP Server (<http://httpd.apache.org/>), an HTTP Web server, and its tools.
- Inkscape (<http://inkscape.org/>), a vector graphics editor.
- KDESvn (<http://kdesvn.alwins-world.de/>), a graphical client for Subversion.
- Kile (<http://kile.sourceforge.net/>), a $\text{\TeX}$ / $\text{\LaTeX}$ editor.

- L^AT_EX (<http://www.latex-project.org/>), a document preparation system and document markup language.
- Linux (<https://www.kernel.org/>), an operating system kernel.
- Maven (<http://maven.apache.org/>), a build automation tool.
- OpenOffice/LibreOffice (<http://www.libreoffice.org/>), an office suite.
- OpenJDK (<http://openjdk.java.net/>), an implementation of the Java SE platform.
- pdftex (<http://www.tug.org/applications/pdftex/>), a T_EX extension.
- Ubuntu (<http://www.ubuntu.com/>), an operating system.
- Subversion (<https://subversion.apache.org/>), a revision control system.
- T_EX (<http://www.ctan.org/tex/>), a typesetting system.
- T_EX Live (<https://www.tug.org/texlive/>), a T_EX distribution.

Short Curriculum Vitae: Simon Mayer

Personal Data

Date of Birth July 16, 1987 in Villach, Austria
Citizenship Austrian

Education

08/2010 – 09/2014 Doctoral student and research assistant in the Distributed Systems Group, Department of Computer Science, ETH Zurich, Switzerland.

Supervision: Prof. Dr. Friedemann Mattern

09/2011 – 08/2014 Studies in Secondary and Higher Education (Swiss Teaching Diploma in Computer Science), ETH Zurich, Switzerland.

09/2010 – 07/2014 Studies of Economics and Business Administration (Focus: Management & Economics), University of Zurich, Switzerland.

Bachelor's Thesis: *Leveraging Information from Online Communities* for Marketing Purposes

09/2005 – 04/2010 Studies of Computer Science (Focus: Distributed Systems), ETH Zurich, Switzerland.

Master's Thesis: *Deployment Support for an Infrastructure for Web-enabled Devices*

Laboratory Project: *Digital Augmentation of Settlers of Catan*

Bachelor's Thesis: *Post-Quantum Key Distribution Schemes relying on General Constraint Graphs*

09/1997 – 06/2005 Grammar School St. Martin, Villach, Austria.