Open Objects

A lightweight and decentralised framework for the Internet of Things

PhD Thesis in Computer Science

Paulo Ricca

Supervision: Kostas Stathis

Royal Holloway University of London

December, 2014

Abstract

This thesis explores the idea of how to use physical objects with digital functionality to interact and communicate with each other in order to assist people with carrying out their everyday activities. The main contribution of this thesis is the proposition of the Open Object concept and its respective supporting framework.

An *Open Object* is a physical object that is capable of network connectivity enabling it to (a) expose and share its capabilities to the outside environment including people and other objects; (b) request the use of capabilities from other objects or external services when needed; and (c) allow people to modify or introduce new functionalities, effectively changing the object's behaviour. This is to be contrasted with existing object or service oriented models whose functionality, although exposed and accessible, is not normally open to changes from the outside.

Basic Open Objects are then used as building blocks to construct assemblies of objects that interact and communicate with each other to manage themselves independently and work as a single entity, which we call *Open Super-Object*. The key issue of Open Super-Objects is how they can be best constructed through the specification of rules that control and coordinate their interactions. This leads to a general approach of architecting, specifying and implementing Open Objects in such a way that the overall coordination of more complex objects is supported by composing the capabilities of simpler objects, regardless of their internal implementation and structure.

Throughout the thesis, Open Objects are presented through a series of scenarios illustrating how the openness of simple and Open Super-Objects can be specified and implemented. However, to investigate the practical value of

Open Objects, we also present and evaluate a prototype implementation of a real-world scenario. In this scenario, an Open Super-Object situated in an office environment notifies people of the availability of a co-worker. We also aggregate a collection of methodologies for implementing different standard computing and software patterns to produce a lightweight Open Object system. We conclude the thesis with a summary of the achievements and a discussion of future work, in particular how Open Objects can be used as a platform to support end-user development.

Acknowledgements

This work could not have been completed without the help and support of a number of people, whom I would like to thank:

Kostas Stathis for all the help, patience and support both at an academic and a personal level. Kostas' knowledge, wisdom, experience and humanistic stance were key throughout the whole study. Also, the rest of the DiceLab, for their invaluable help and feedback.

My parents for unconditional support, friendship, love, kindness and for being all-round good examples, for shaping who I am today. Also my sister and my nieces for bringing me so many smiles and laughs.

José Danado for being my informal yet involuntary tutor prior to these studies.

Eduardo Dias and Nuno Correia for the great help and support during the PhD proposal stages.

Joana Gomes for the friendship and for that dinner where she convincing me to start this adventure, in the first place.

Frieder Ganz who was involved in the creation of one of the case-studies — the lucky waving cat — and for being an all-round good friend.

Ulli Schaechtle for being a great friend and for helping with reviewing part of the thesis.

My little surrogate sister Olivia Vass for her help with English, for all the cups of tea and for being the best cooking partner in the world.

Shahi Ghani for being my little brother, a good friend and providing a great work environment.

Anna Downey for the support, endless patience for my occasional grumpiness and for being the best partner in crime.

And Anastasia Ushakova for all the love, motivation and general banter during the writing of the thesis, and for laughing at my bad jokes.

Also, I would like to symbolically thank all the pasta and wine and laughs for helping me go through any frustrating moments.

All of you, one way or another changed my life for the better during (and beyond) the duration of this study.

Contents

1 Introduction **3**

1.1 Motivation . . . 5

1.2 Aims and Objectives . . . 7

1.3 Contributions . . . 8

1.4 Structure of the Thesis . . . 9

1.5 Previous Publications . . . 9

2 Background **11**

2.1 Ubiquitous Computing . . . 12

2.1.1 Ambient Intelligence . . . 12

2.1.2 Internet of Things . . . 13

2.1.3 Web Of Things . . . 13

2.1.4 Sensor Networks . . . 14

2.2 Smart Objects . . . 14

2.2.1 Computational Capabilities or “Smart” vs “Dumb” objects 15

2.2.2 Connectivity . . . 15

2.2.3 Interoperability . . . 15

2.2.4 Autonomy . . . 16

2.2.5 Physical vs Virtual . . . 16

2.3 End-User Development on Smart Objects . . . 16

2.3.1 Visual Programming . . . 17

2.3.2 Programming by Example . . . 18

2.3.3 Sketching . . . 20

2.3.4 Meta-Design . . . 21

2.4 Internet of Things Platforms . . . 21

2.4.1 Requirements for an open development framework for IoT 22
2.5 Ad-Hoc Interactions 25
2.6 Summary 25

3 Open Objects 27
3.1 An Environment of Open Objects 29
3.2 Baby Monitor Scenario 34
3.3 Open Object Architecture 35
3.3.1 Capabilities 35
3.3.2 Events 37
3.3.3 Behaviours 37
3.4 System Behaviours 39
3.4.1 Behaviour Mapping 39
3.4.2 Rule Execution 40
3.4.3 Rule Repository 40
3.4.4 Event Broker 40
3.4.5 Behaviour Advertising 40
3.5 An overview of the framework 41
3.6 Interactions between System Behaviours 42
3.6.1 Event-based Interaction 43
3.6.2 Behaviour Registration 44
3.6.3 Behaviour Discovery 44
3.6.4 Rule Composition 45
3.6.5 Rule Execution 45
3.7 A Complete Open Object 46
3.8 Open Super-Objects 47
3.9 Summary 48

4 Open Object Description Language: Specification and Implementation 51
4.1 Our Specification Approach 51
4.2 Open Object 53
4.2.1 Open Super-Object 54
4.3 Capabilities 54
4.4 Events 57

4.5 Behaviours 62
4.6 Rules 64
4.7 Summary 65

5 Implementation 67
5.1 A few notes on Methodology and Notation 68
5.2 Behaviour Advertising 69
5.2.1 Internal State 69
5.2.2 Events 69
5.2.3 Capabilities 70
5.3 Behaviour Mapping 71
5.3.1 Internal State 71
5.3.2 Events 72
5.3.3 Capabilities 72
5.4 Event Broker 73
5.4.1 Internal State 74
5.4.2 Capabilities 75
5.5 Rule Repository 75
5.5.1 Internal State 76
5.5.2 Capabilities 76
5.6 Rule Execution 77
5.6.1 Events 77
5.6.2 Capabilities 77
5.7 A RESTful approach 79
5.7.1 Capabilities as Resources 82
5.7.2 Web Client and Server as Actuator and Sensor 84
5.8 Experimental Implementation Platforms 84
5.9 Summary 86

6 Case-Studies 87
6.1 Smart Lucky Waving Cat 87
6.1.1 Methodologies 88
6.1.2 Results and Conclusions 89
6.2 Algorithmic Test 89
6.2.1 Methodologies 89

6.2.2 Results and Conclusions 92
6.3 Open Power Socket 95
6.3.1 Methodologies 96
6.3.2 Results and Conclusions 98
6.4 Resilience Test 99
6.4.1 Methodologies 99
6.4.2 Results and Conclusions 101
6.5 Office Availability Device 101
6.5.1 Methodologies 102
6.5.2 Results and Conclusions 106
6.6 Adding a new Behaviour to an Object 108
6.7 Conclusions 113

7 Evaluation 115
7.1 Status 115
7.2 Comparative Evaluation 119
7.3 Summary 123

8 Conclusions 125
8.1 Review and Discussion of the Achievements 125
8.2 Future Work 127
8.2.1 End-User Development Tool 128
8.2.2 Maturing the Platform 128
8.2.3 On Predicates First vs Context First when designing an End-User Development language 128

Appendices 141

A Open Object System Behaviours: Capability Algorithms 143

List of Figures

1.1 Two examples of RFID tags 4

2.1 Service Creation Kit in m:Ciudad 18
2.2 A data flow visual programming interface. 19
2.3 Scratch Visual Programming Interface. 19
2.4 An Arduino microcontroller. 20

3.1 Objects in physical and digital Environments 28
3.2 A "smartness" scale of objects. 32
3.3 Open Object Architecture 35
3.4 Open Object life-cycle 36
3.5 The global Open Object Ontology. 41
3.6 The scope of the Open Objects framework. 42
3.7 The event passing process. 43
3.8 Behaviour Registration 44
3.9 Behaviour Discovery 44
3.10 Rule Composition 45
3.11 Rule Execution 45
3.12 A Complete Open-Object 47
3.13 Incomplete object sharing System Behaviours 48
3.14 Open Super-Object diagram 49

5.1 Bottom-up execution 80
5.2 Top-down execution 80
5.3 Another Top-down execution example 81

6.1 Smart Lucky Waving Cat 88
6.2 Open Power Socket 95
6.3 Open Door Sign 102
6.4 Open Cube Controller 102

Glossary

AmI Ambient Intelligence. 12

ECA Event-Condition-Action Rules. 65, 116, 120

EUD End-User Development. 16, 17, 20, 22, 116, 120, 121, 126

HDL Hardware Description Language. 29

IoT Internet of Things. 11–14, 20, 22, 24

OODL Open Object Description Language. 51, 66

OOP Object-Oriented Programming. 68

OSH Open Source Hardware. 29, 30

OSS Open Source Software. 29, 30

PaaS Platform as a Service. 23

PCB Printed Circuit Board. 29

RSDL RESTful Service Description Language. 122

SMC Self-Managed Cells. 24

WoT Web of Things. 13

WSDL Web Service Description Language. 122

WSN Wireless Sensor Network. 14

List of Procedures

5.1 Request System Behaviour's Capability — Expanded Procedure 69
5.2 Request System Behaviour's Capability — Simplified Procedure . 69
5.3 Behaviour Mapping — Get Behaviour URIs 73
5.4 Event Broker — Forward Event 75
5.5 Rule Repository — Get Rules By Event 76
5.6 Rule Execution — Execute 77
A.1 Behaviour Mapping — Register Behaviour 144
A.2 Behaviour Mapping — Deregister Behaviour 144
A.3 Behaviour Mapping — Activate Behaviour 144
A.4 Behaviour Mapping — Deactivate Behaviour 145
A.5 Behaviour Mapping — Rank 145
A.6 Behaviour Mapping — Get Behaviour URIs 145
A.7 Rule Execution — Execute 146
A.8 Rule Repository — Register Rule 146
A.9 Rule Repository — Deregister Rule 147
A.10 Rule Repository — Activate Rule 147
A.11 Rule Repository — Deactivate Rule 147
A.12 Rule Repository — Get Rules By Event 147
A.13 Rule Repository — Get Rule 148
A.14 Event Broker — Forward Event 149
A.15 Event Broker — Subscribe 150
A.16 Event Broker — Unsubscribe 150
A.17 Behaviour Advertising — Register Behaviour URI 150
A.18 Behaviour Advertising — Register Behaviour 150
A.19 Behaviour Advertising — Advertise 150

1 Introduction

As technological advances produce smaller and cheaper electronics, more everyday physical objects become embedded with computation, and the boundaries of the definition of a computer start fading. RFID tags [Wan06] (pictured in Figure 1.1), for instance, can be thought of as minuscule computers capable of processing information, interacting with sensors and communicating wirelessly without batteries. We are already surrounded by these small computers in our everyday lives. They are glued to cereal boxes, tagged onto clothes, put into books and library cards, and they are virtually free. This kind of computing was explored by Mark Weiser, who conceptualised ubiquitous computing [Wei91], as disposable computing. If Moore's Law remains valid during the foreseeable future [TP06], soon most everyday objects will contain a form of embedded electronics capable of processing information from the environment, actuating through the use of the object's capabilities and communicating with each other wirelessly.

From the point of view of an object manufacturer, at a low extra production

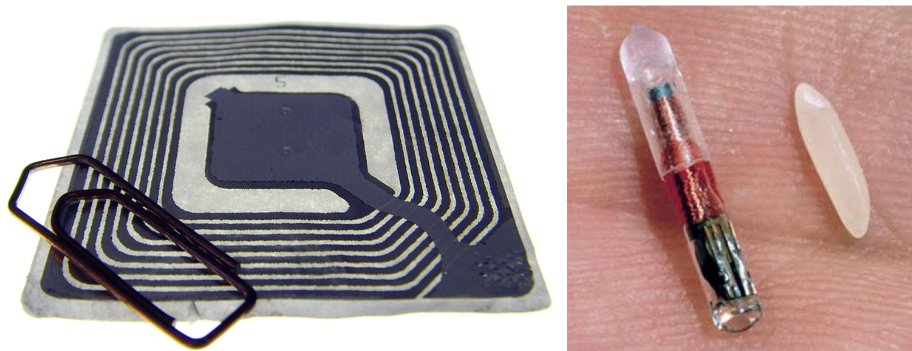

Figure 1.1: Two examples of RFID tags. On the left, the tag commonly found in DVD boxes or books, on the right, a tag that identifies pets, comparable to a grain of rice in terms of size

cost, we can imagine advantageous scenarios where hardware companies produce domestic appliances that are able to send back anonymous usage information or malfunction reports while being used by the end-user as with computer software. Remote assistance or firmware updates could also be made possible with an Internet connected appliance. There are many examples [HDK08, CSDC11, FW99] of physical objects having an increased value to the user by means of a thin layer of computation built onto it, and the "killer applications" are probably yet to be thought of. While we definitely see the advantages of the applications from the object's producers' point of view, exactly how to make use of this computation from the users' point of view remains less clear.

The field of Ambient Intelligence often focuses on applications where interfaces and behaviour of systems adapt according to context [AM07]. A recurring scenario [GHL05, HMF+08, DBS+01] depicts a smart kitchen that can sense, among other things, what ingredients are present in the fridge and cupboard. A user might look up recipes based on this information and once one is chosen, the oven might set up itself according to the settings specified by the recipe. Examples such as this one rely on heterogeneous objects and services working together to achieve a common goal that fits the current user's needs. Suitable frameworks and protocols need to be developed to allow for scenarios such as this to be possible.

In some cases, decentralisation and support for serendipitous collaboration can bring interesting advantages. A wheelchair user may, for instance, configure

the functionality of the chair so that it requests traffic lights to stay green for a longer time, in order to give the person extra time when crossing roads. Another user who enjoys waking up to his favourite radio station may define a rule on his mobile phone that turns on his radio at the time the alarm is set to, and this rule may also carry on working if the user is in a hotel, for instance, where the mobile phone requests the hotel tv to turn on to the same radio station, at the right time. In a further example, a decentralised ad-hoc connection of cars in a motorway [BHBR10] might make it possible for cars to inform drivers of accidents ahead or ambulances behind. Decentralisation and unpredictable opportunities for collaboration bring different opportunities but also new challenges that foster new research around these areas.

1.1 Motivation

Our interest in these kinds of applications had previously drawn us into imagining how End-User Development (EUD) could assist users modelling their own rules that solve concrete problems, unique to their personal lives, through the use of their existing heterogeneous physical objects. During initial research we found that not only there were no existing software frameworks to support these kinds of applications, but also that there were little or non-existing generic and consistent frameworks of thought to be based on. We therefore shifted our interest into producing a working framework for the class of problems that emerge from these types of scenarios where collaborations between physical objects take advantage of not only decentralisation and serendipitous interactions, but also from the final user being able to specify exactly how the collaboration occurs, and embed these specifications of new functionalities into these objects.

The theoretical framework, the technology and the standards are now at a point where we can have truly useful ubiquitous computing environments. This thesis envisions a near future where physical objects with a thin computational layer have the ability to (a) expose their capabilities to the outside, sharing them with other objects; (b) outsource certain capabilities from other objects or external services when needed; (c) allow their users to modify and attach functionalities, effectively changing their behaviour. This vision encompasses several aspects of openness. We believe that sharing and openness in general is a catalyst for innovative solutions; therefore it was our intention to design

a framework with openness at its core. We envision scenarios where end-users are able to introduce new and custom functionalities and capabilities into their existing objects or combine multiple objects to form a more complex functionality. We believe that this process should become collaborative: a crowd sourced functionality store of open source functionalities for different objects, where users could mix and match functionalities designed by others or modify them to target their personal needs.

The other perspective on openness is the already mentioned outsourcing of capabilities by the objects: instead of an object being constrained by its capabilities it can focus on what it does best and leave other accessory tasks to other objects or external services that may be more specialised in them.

A simple example, a kitchen oven's main function is not time keeping, yet it makes use of a clock, thus needing an extra hardware interface for it. Alternatively, if the same oven was able to outsource this capability from a nearby clock or a web service, it could same resources and have a more streamlined physical user interface that focuses on the oven's main task. This would also make the user's life easier during power failures or daylight saving time changes. Another example could be a bread maker that schedules its time according to the user's alarm clock. There are numerous other scenarios, not only in a domestic environment but also in industrial and public ones, where objects sharing their capabilities and letting users build a layer of functionality on top of them can bring great advantages.

In the process of materialising this vision, a series of challenges arise. What kind of technologies would be more suitable to make it possible to seamlessly connect previously unrelated objects? What kind of language would be simple yet powerful enough to allow end-users to create new distributed systems with enough complexity to fulfil their needs? What kind of framework and architecture would suit such decentralised, pervasive, dynamic and unstable environment? How could the creation of new functionality of an object be turned into a fun, collaborative and open development experience? There is an increasing need for distributed systems frameworks that allows these types of objects to communicate and cooperate in an effective way independently of their limitations.

1.2 Aims and Objectives

We aim at making technology work beside us, not instead of us. We believe that smarter everyday-life technology is more than just smarter physical objects. It's about creating smarter object interactions and generally making our life simpler. It's about sharing and connecting what already works well in order to build something more useful and powerful.

We intend to give back control to the users over their devices by promoting openness, transparency and innovation in the design and development of functionalities and behaviours of everyday objects. We aim at releasing device owners from unwanted design decisions that compromise the best usage of their equipment. We aim at making it possible to continue the design and development of an object, after its purchase: Object designers create objects in a way that allows these to be modified, expanded and connected to better target users uniquely personal needs. This concept is often referred to as Meta-Design [FGY+04].

One of the ways to fulfil these aims is to investigate how we can take a computationally capable physical object and take it one step further into being able to not only share its internal capabilities with other object's, people or remote services, but also to outsource capabilities that it does not possess or even to let new capabilities be easily programmed and embedded by an end-user. We intend to make this extra step as small as possible in order to allow minimally thin and cheap implementations, while still supporting complex scenarios where necessary.

Technically, when physical objects become computationally capable and are able to connect to Internet services, it is said that we are dealing with an Internet of Things (IoT) [AIM10] environment. Most existing approaches to the Internet of Things take the weight off the objects and devices themselves by transferring application logic and heavier computation to a centralised machine. This approach makes sense when heavy, real-time processes are concerned but there are many situations when processing power, processing time and memory usage are not critical, but other constrains such as unstable or ad-hoc communications, security or privacy concerns make it preferable to have objects performing certain tasks internally. Furthermore, we could benefit from hybrid systems where some tasks are carried out by small objects and other heavier tasks by exter-

nal, specialised services. For such highly decentralised system to work we need to find a minimum common denominator, a common framework that is light enough to be implemented on very lightweight objects but flexible and powerful enough to support a very wide range of possible applications.

With these aims in mind, we outline the following set of objectives:

- Develop an architecture for a software layer that allows physical objects to expose their capabilities to the environment in which they are situated as well as outsource capabilities to other objects that support them and can provide them;
- Study the features of a rule-based framework to describe internal functionalities and coordination of objects, which in turn can be used as a target platform for an End-User Development Layer;
- Propose a specification for Behaviour, Capability and Rule descriptions;
- Develop a functional and reusable implementation of the architecture and the rule-based framework.

1.3 Contributions

Throughout our work, we introduce our main contribution — the concept of Open Objects. These are physical or emulated artefacts, whose embedded computing allows them to share and/or outsource resources, such as sensors, actuators or processing capabilities while also allowing the user to modify or augment its internal functionalities. In order to support this paradigm, we describe a lightweight, distributed framework aimed at user-centric orchestration of Open Object on ambient intelligence environments, and demonstrate an implementation and a number of case-studies. One of the innovative aspects of this framework are the methods of decentralisation that allow for partial implementation of the framework on lightweight devices, which can then be treated as "first-class citizens", instead of lower level components accessible through proxies. Another important contribution of this work is the rule syntax and engine, which support ad-hoc interactions and high levels of expressiveness in terms of functionality description. Open Object Rules combine, in the same syntax and environment, interactions between system management functionalities (System

Behaviours) and application level ones (Domain Dependent and Independent Behaviours), and are also used to define new capabilities that can be added to existing objects — thus supporting the concept of Meta-Design explored in Chapter 2. The design of these rules was made with the intention of supporting End-User Development by using a language that can be closely mapped into such a tool.

1.4 Structure of the Thesis

In Chapter 2 *Background* of this thesis, we lay the ground for the rest of the research, contextualising the work from the points of view of Ubiquitous Computing, Distributed Computing and End-User Development. Chapter 3 *Open Objects* introduces the core concepts behind the Open Objects framework and how they contribute to answering the questions raised in the previous chapter. In Chapter 4 *Specification*, we formalise the Open Objects framework and its data formats, laying the groundwork for implementation. In Chapter 5 *Implementation* we present our implementation of the Open Objects framework, a number of implementation issues and the system management layer of the framework. Chapter 6 *Case-Studies* presents a number of experimental prototypes, where the previously presented implementation is evaluated in controlled and real-world scenarios. In Chapter 7 *Evaluation*, we analyse and discuss the results of the case-studies, the current state of the implementation and a systematic comparison with a number of other platforms. Finally, in Chapter 8 *Conclusions* we provide a summary of our contributions and lay the ground for possible future work.

1.5 Previous Publications

Below are some of our publications that have directly contributed this thesis.

- Danado, J., Davies, M., **Ricca, P.**, and Fensel, A. *"An authoring tool for user generated mobile services."* In Future Internet-FIS 2010. Springer Berlin Heidelberg, 2010. 118-127.

  The participation in this project sparked the motivation behind this thesis.

- **Ricca, P.**, Stathis, K., and Peach, N. *"A lightweight service registry for*

unstable ad-hoc networks." In Ambient Intelligence. Springer Berlin Heidelberg, 2011. 136-140.

This publication reflects part of our involvement with an Ambient Intelligence project focused on decentralised and ad-hoc interactions in unstable and/or unreliable networks (namely on war scenarios). This involvement resulted partially in the implementation of the Event Broker and the Behaviour Mapping, important components of the framework, detailed in Chapter 5.

- **Ricca, P.**, and Stathis, K. *"Open Objects for ambient intelligence."* In Ambient Intelligence. Springer Berlin Heidelberg, 2012, 320–327.

  In this publication we outlined the basic concepts behind the framework. These concepts matured and originated Chapter 4.

- **Ricca, P.**, and Stathis, K. *"A RESTful and decentralised implementation of Open Objects."* In Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing. ACM, 2013.

  Presented in this paper is an early stage of the implementation of the framework and a brief description of one of the case-studies in Chapter 6.

2 Background

This thesis lies at the intersection of a range of different areas. Research under the umbrella of Ubiquitous Computing has provided insight into how computers and human-computer interaction will be likely to evolve and how it may shape our life as individuals and as a society. Ubiquitous Computing is commonly referred to as the type of computing that fades into the physical background and becomes invisible to the user, frequently focusing on the physical aspect of the human-computer interactions. Ambient Intelligence is a related field that can be thought of as the branch of Ubiquitous Computing that is concerned with intelligent physical spaces that identify users activities and adapt their behaviour accordingly. The Internet of Things (IoT) usually offers a more technical and practical point of view on how physical objects can be designed or transformed to be able to connect to the Internet, what implications arise from this connection and what infrastructures and standards should be adopted to support these ecosystems.

As often happens, similar problems are tackled from adjacent areas of re-

search, from different perspectives — the field of Distributed Computing deals with matters that arise when computation tasks are split between different machines, issues which also affect IoT applications frequently. The Service-Oriented Computing field offers a work base for providing physical objects with means of sharing their capabilities and connecting to other objects. The Web of Things ideas suggest the use of existing Web tools and protocols to build the IoT around physical objects with service-oriented architectures. Over the next sections, we have a closer look over each of these areas of interest and research.

2.1 Ubiquitous Computing

Ubiquitous or Pervasive Computing was a term coined by Mark Weiser [Wei91] from Xerox Corporation to illustrate the vision of an environment with an invisible, unobtrusive, helpful and intelligent grid of small computers that surround the user, focused on helping him on daily tasks by predicting future actions and understanding social contexts. Ubiquitous Computing has its origins on the intersection of two apparently incompatible trends. On one hand, computers and technological gadgets that are as unobtrusive and invisible as possible are sought after — the best user interface is the self-effacing one — and on the other hand the amount of information available to and desired by us is exploding. This paradigm tries to make information, intelligence and control easily accessible to the user, without adding more entropy to its already technologically complex everyday life.

There are several schools of thought in Ubiquitous Computing, each of them focusing on different aspects of the same problem/environment. In the next subsections, some of these points of view which are critical to the present research are presented.

2.1.1 Ambient Intelligence

The vision of AmI [ZEBD98, SS09] is to create environments that sense, anticipate and respond to people's actions. It can be considered a point-of-view over the Ubiquitous Computing paradigm, closer to commercial services and industrial products, perhaps due to its industrial Philips roots. Ambient Intelligence applications often deal with larger scenarios than the ones we are interested in, such as smart buildings and adaptive power grids. Furthermore,

their philosophical approach is often antipodal to ours as it commonly focuses on autonomically acting on behalf of, or supporting the user by recognising its activities[CAJ09, STF07, DHC05]. Our approach is lower level, focused on providing the necessary tools and infrastructures to empower the user to let them define how environments and objects should act on different circumstances.

2.1.2 Internet of Things

As a branch of the distributed computing research topic, the Internet of Things [AIM10] is a concept which explores the broader Ubiquitous Computing paradigm in what relates to the way small heterogenous computers and smart objects [KKFS10] can connect to each other, coordinate, access and share information to the user through loosely coupled services. The concept became more relevant when it was indicated[1 2] that the number of things (the word "things" is commonly used to describe electronic artefacts with embedded systems that are either too simple or too small to be called computers, which are able to sense and / or actuate on the environment) became larger than the number of people connected to the Internet, which means that we have to rethink how the two layers can interact and overlap in order to create meaningful and useful connections.

2.1.3 Web Of Things

The Web of Things [GT09] (WoT) concept is based on the idea that, assuming the IoT networking challenges are already mostly solved, the research should further move itself to the upper layers of the system, namely the messaging protocols to be used. In other words, we can think of the WoT as a layer over the IoT transport layer, a parallel to how the Web protocols and applications are a layer over the Internet protocols. The WoT ideology intends to reuse as many of the proven Web technologies as possible to suit the IoT objective, namely, HTTP and REST [Pas12] for message handling and Atom syndication for service discovery. As the IoT frequently deals with devices with limited processing power and constrained connectivity, it's useful to optimise the transmissions as much as possible. With REST we use the actual HTTP headers as meaningful message envelopes instead of just being generic network wrappers for another

[1] http://www.gartner.com/newsroom/id/2636073

[2] http://www.theconnectivist.com/2014/05/infographic-the-growth-of-the-internet-of-things/

messaging protocol, therefore minimising and simplifying the way devices share data.

2.1.4 Sensor Networks

Wireless Sensor Networks (WSN's) [SGAP00] are networks of specially distributed autonomous devices that sense a particular aspect of a given environments and collectively create a geographical map of data which is then processed or interpreted by a central processing unit. Common applications of WSN's are sensors that monitor building's structural integrity, agricultural fields, wastewater monitoring, landslide/earthquake/forest fire detection and industrial plant monitoring. Challenges in WSN's are usually related to communication, trust issues, spread of information through grids of sensors, ad-hoc communications and dynamic/unstable networks [CK03, SKPB07]. Although this area of research is not our direct interest, it can be seen as one of the possible applications of the outcome of the project.

2.2 Smart Objects

Making physical objects "smarter" or more cooperative is hardly a novelty. Through the last decades, several research and industrial fields have worked towards that aim. Sensor Networks gather data from lower level autonomous sensors to make it accessible over an aggregating centralised system (e.g. [KEW02, GBCM11]). Embedded Electronics have evolved so that small scale electronics can be embedded into physical objects at very low cost enabling cheap "intelligence" on everyday items (e.g. [FW99, NRL07]). The fields of the IoT and Ubiquitous Computing aim at integrating physical objects into computer networks merging the user's digital and physical environments [KKFS10].

Despite this variety of fields (or perhaps because of it) there is, however, no consensus over the terminology used in these contexts. A number of questions seem to have no consistent answer: Are *Things* and *Objects* the same thing? What does it mean to be "Smart"? Do they encompass only objects with a physical presence? Are they a virtual representation of physical objects or the objects themselves? Do they need to have any form of computational capabilities embedded into them? Do they need to be able to connect to a computer network?

2.2.1 Computational Capabilities or "Smart" vs "Dumb" objects

Objects that have no computational capabilities but have a barcode [QBVG08] or an RFID [FW99] attached, for instance, are often referred to as "Dumb" objects but are still connectable or identifiable through appropriate automated or semi-automated scanners. When a physical object is embedded with digital electronics (which become its computational capabilities), it is said to become "Smart". It is, however, unclear what kind of computation and how much processing power is required for an object to be considered "Smart".

2.2.2 Connectivity

An object may possess computational capabilities but no connectivity to other objects, only being capable of interacting with its own sensors and actuators. In that case, it is discussable if the object is to be classified as "Smart" or not, and literature is ambiguous. One of the main issues of connectivity is power consumption — as a general rule of thumb, the longer the distance of communication, the high the power consumption. On the lower end of the consumption scale we have RFID and, NFC, in the middle we have bluetooth and at the top of the scale Wi-Fi. Battery-powered devices adopt different strategies to preserving power while maintaining connectivity. Some of these strategies imply becoming disconnected for periods of time, which poses a challenge for interoperable systems.

There has been recent developments regarding application communication protocols, such as CoAp [CoA12] that targets small, low-powered devices. This protocol is characterised by having low overhead (compared to HTTP) and support for multicast, both important for the kind of scenarios involving low-powered smart objects.

At a lower level of the OSI model, 6LowPAN [Mul07] allows transmission at lower data rates through a variety of physical means, and is also targeted at small, low-powered devices.

2.2.3 Interoperability

Some objects with connectivity capabilities may be able to interact and collaborate with other objects. Interoperability is key in this context due to the diversity of hardware and software platforms. Due to this diversity, we can

observe a trend of Service Oriented Architectures being applied in this context [AIM10, SKG+09, IGH+11, HKS09], due to its implementation independency nature. It is unclear, however, whether or not interoperability is implied when referring to a physical object as "Smart".

A common issue regarding interoperability is how to choose the operational semantics of the interactions, specifically on decentralised scenarios in the cases where system nodes do not have access to global ontologies.

2.2.4 Autonomy

An object is autonomous when it is able to operate as intended on its own, without having its actions controlled externally. Objects are not autonomous when they behave as sensors or actuators accessed or controlled by other objects or nodes in a network.

2.2.5 Physical vs Virtual

Another point of discordance is related to the possibility of having virtual objects as representations of physical objects, groups of objects, or services. Conceptually, a Virtual Smart Object is different from a Physical Smart Object as it is not directly attached to a physical entity, but in practice both types would behave and interact similarly.

2.3 End-User Development on Smart Objects

We became interested in the areas of study mentioned above from the participation in a project called m:Ciudad [DDRF10]. This project focused on enabling both the development and the execution of small applications on mobile devices. Users could modify or create these small mobile applications on their mobile devices from scratch using a block-based EUD tool (example of the Service Creation Kit, the EUD tool, in Figure 2.1). This project showed it was feasible to implement the tools necessary to allow the common user to construct powerful systems in a simple way. This conclusion, partly inspired by [CFMD10], together with the uprise of cheap DIY electronic platforms (such as Arduino[3] or Rasperry Pi[4]), inspired us to think about a future world where the

[3] http://www.arduino.cc

[4] http://www.raspberrypi.org/

common user is given back control over their own physical objects. By combining their functionalities with physical mashups [HDK08], connecting them to the Internet and modifying their behaviour, we could make the better suit the user's personal life.

Throughout the past decades, efforts have shifted from making systems easy to use into making them easy to develop [LPKW06]. We believe that the Web 2.0 revolution, where content consumers became content creators, mostly happened simply because the right tools (such as Blogging platforms and general purpose Content Management Systems) were imagined and developed. We also believe that the next step will be the evolution from content creators to functionality creators. Tools (such as Yahoo Pipes[5], IFTTT[6], Zapier[7] or Tasker[8]) that allow common users to create custom functionalities and system rules are already becoming mainstream.

End-User Development (EUD) is a field of research which focuses on developing techniques and tools that allow people without any specific programming skills to be able to create or modify software. EUD plays an important role by putting part of the innovation in the end-user's hands, opening new opportunities for customisation and amplification. A notable example of EUD is the use of spreadsheets in which users may create highly complex applications with apparently little complexity based on simple rules and mathematical formulas.

EUD makes it possible for users to closely participate in the design of the systems they use, making them fit better to their own specific tasks, rendering either their professional or personal lives more satisfactory by spending less time on repetitive tasks (that are better performed by well designed computational systems) and focus on what humans do best. In order to enable software development by end-users several methodologies have been created and studied. Some of the more significant and influential ones are shortly described below.

2.3.1 Visual Programming

A Visual Programming Language is one that users manipulate graphic representations of software development components, instead of writing program

[5] https://pipes.yahoo.com/pipes/
[6] https://ifttt.com/
[7] https://zapier.com/
[8] http://tasker.dinglisch.net/

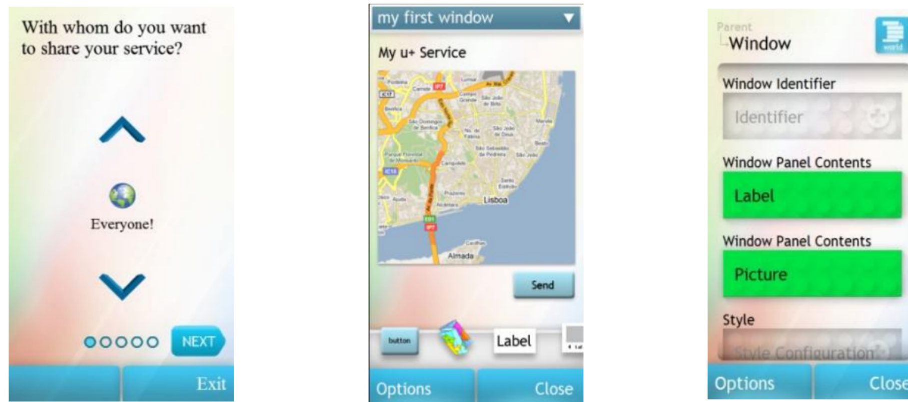

Figure 2.1: Service Creation Kit in m:Ciudad. The user is able develop a mobile app on a mobile device using the SCK. Pictured are three different levels of increasing complexity which complement each other.

code [Mye90]. This offers several advantages to beginners and non-programmers such as less memorisation, being less prone to syntactic errors and letting users organise the program logic in a visual way that makes sense to them (such as the example picture in Figure 2.2). Although there are some variations based on similar principals, the user interfaces commonly used for these languages are composed of functional blocks with inputs and/or outputs, and arrows or lines connecting the blocks which represent logical sequences(or workflows), data flows or control flows. There are also examples of hybrid approaches, such as Scratch [RMMH+09] (Figure 2.3), a tool developed to assist young learners when learning to programme, which wraps common procedural programming metaphors in a graphical presentation.

2.3.2 Programming by Example

When programming by example, instead of implementing a certain functionality directly, the user performs the action itself on one or various particular use cases and it's up to the underlying system to infer the generalised logic behind the operation [Hal84]. This is a practical way of automating repetitive tasks and is used, for example, in the field of Robotics to program robots. A clear advantage of using this technique is that users use the same user interface to use the system and to program it which allows them to work on a familiar environment instead of having to learn a new one.

Figure 2.2: A data flow visual programming interface.

Figure 2.3: Scratch Visual Programming Interface.

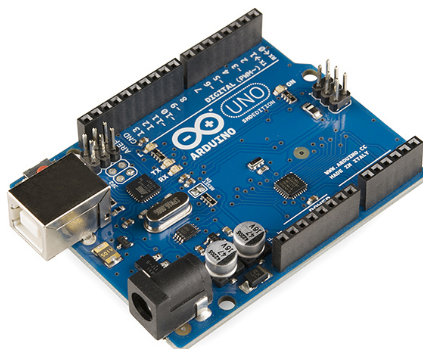

Figure 2.4: An Arduino microcontroller.

2.3.3 Sketching

Half-way between conventional programming and EUD sits Sketching, a term used when referring to small software programs, created without taking into account good software architecture practices or solid longevity. Sketches are often written by non traditional programmers such as visual or sound artists usually to be used just by themselves or a small group of interacting people. The sketching environments, either written in a traditional programming language or a subset of one, such as Java (Processing[9]) or C (Arduino[10] [Gib10]), or in a visual programming manner(such as Pure Data[11] or vvvv[12]), usually aim at being simple to use for beginners but still allow a great level of control and complexity for advanced programmers. Sketching is also used when someone wants to prototype functional systems quickly.

Arduino (pictured in Figure 2.4) is a rapid hardware prototyping platform that is commonly used by artists and “DIYers” which allows them to create custom, very focused ad-hoc electronic artefacts with great ease, relatively low production costs and, perhaps more importantly, with in-depth knowledge of electronics and/or embedded programming. For these reasons, Arduino became a very good platform for implementing and testing IoT environments with-

[9] http://www.processing.org
[10] http://www.arduino.cc
[11] http://www.puredata.info
[12] http://www.vvvv.org

out having to study complicated electronics, while still keeping the prototyping costs down. Technologically, the Arduino is the combination of a small and simple to use prototyping board with an embedded micro-controller, a boot loader software programmed into the micro-controller which makes it easy to programme the devices, a development environment based on the Processing environment and a vast amount of open source libraries that facilitate the usage of the most common sensors, actuators, integrated circuits and shields (plug-and-play boards that provide additional functionalities such as motor control, ethernet or liquid crystal displays). Software running inside the Arduino is usually written in the form of small Sketches, programmed in a simple to use Processing-based IDE which compiles directly into the Arduino by connecting to a USB port. That and the very supporting community makes it one of the easiest electronics platforms to develop to.

2.3.4 Meta-Design

The concept of Meta-Design [FGY+04] is based on the idea that instead of the design and functionality of a hardware or software artefact being finished and closed when acquired by the final user, it is open allowing the user to further improve or modify the way it operates. Typically, the end-user would have access to an End-User Development tool that is able to interpret and rewrite the functionalities of the artefact. A common (although not ideal in the sense that it requires some knowledge of computer programming) example of Meta-Design is the ability to create *Macros* that alter or expand the operation of pieces of software, i.e. Visual Basic Macros in Microsoft Word.

2.4 Internet of Things Platforms

Motivated by the points explained earlier, we decided to further investigate the existence of an appropriate technical framework to support EUD over applications compromised of a group of Smart Objects. As we are also interested in exploring the usage of external services over the Internet, frameworks built to support IoT applications were obvious candidates, although we also looked into multi-agent systems platforms, which also deal with similar challenges.

2.4.1 Requirements for an open development framework for IoT

During our investigation we defined six characteristics to look for in a possible framework:

- **EUD support** from the ground-up. Arguably, the ability to support EUD and give users control over their objects is key in the IoT, hence we intend to find a framework that is preferably designed from the ground-up to support it.
- **Ad-hoc interactions**. We do not assume that the user will predict all the use-cases and environments where its objects may operate, or with which other objects they are going to interact with, therefore it is important for the framework to be able to handle ad-hoc situations.
- **Decentralised architecture**. Surveys have shown [SK11] (and supported by our own informal interviews) that there is a general public concern about the risks involved with delegating the control of a house to an external organisation over the Internet. Whether or not the concern is based on real risks is arguable, but there are significant advantages in having a decentralised and local (possible to implement between devices on a local network) architecture, for instance the support for ad-hoc interactions and the non dependency of an available Internet connection (i.e. the house system does not stop if there is no Internet connection). Moreover, centralisation can be seen as a particular case of decentralisation, therefore, if a system supports the latter, the former is also supported.
- The **Lightweightness** of the framework is key to support the previous two point considering the fact that many of the devices used in the envisioned interactions would be low-powered and/or have low processing and memory capabilities.
- **Interoperability** with other systems. As there are many frameworks and platforms, it is important that different systems are able to communicate and interoperate between each other, preferably through established standards.
- It is believed [FG06, Ehn08] that **Meta-Design** has the potential of significantly increasing the usefulness and flexibility of objects, therefore, we find that allowing the user to modify the functionalities and operations

of an object is an important condition to better support the envisioned scenarios.

With these characteristics in mind, we assessed a number of platforms and frameworks, of which we chose a selection for a brief discussion. Although this selection is very heterogeneous and not directly comparable (in some cases they are complimentary), there is enough overlap of functionality, hence the choice to evaluate the platforms together. This way we cover the different aspects of the Open Objects framework feature by feature.

Contiki[13] [DGV04] is a very lightweight and highly efficient operating system with a very small memory footprint (around 40kb), built to run on low-powered devices. It supports several communication protocols and has ports for many different hardware platforms, making it a good platform to support enterprise heterogenous applications. Being an operating system, it is meant to be used as a platform onto which other applications are developed, therefore it is a layer below the level of framework we aim at using (which could potentially be running over Contiki). An interesting feature of Contiki is that it allows the developer to run the system in simulation mode, making it possible to observe the behaviour of applications in a closed software environment.

ThingSpeak, **EVRYTHNG**[14] and **Xively** are among a group of cloud based Platform as a Service (PaaS)[Law08] providers that are specialised in aggregating and centralising information and data from existing Internet connected devices. Both ThingSpeak and EVRYTHNG seem to be directed at individual hackers and developers, and the latter also provides custom advertising campaign solutions. Xively offers similar features but is directed at enterprise applications. The three provide a comprehensive and standard layer for interfacing with applications for the IoT based on Web technologies but they do not provide an application layer.

ioBridge provides simple to use hardware platforms for connecting existing hardware devices to the Internet by sending data to a centralised server, which in turn makes it accessible through a RESTful interface. ioBridge is mostly targeted at hackers and hobbyists that want to take control or modify their existing objects.

[13]http://www.contiki-os.org/
[14]https://evrythng.com

HomeOS [RSL+04] is an operating system developed by Microsoft Research targeted at home automation. It runs on a common computer and discovers/connects devices in a local networks. It provided to developers high-level abstractions to orchestrate the devices in the home. An interesting feature is the way the system handles the external devices using the familiar computer periphericals metaphor. **Lab of Things** is framework (mostly targetted at researchers) built on HomeOS that facilitates the creation of applications on this system. The framework has a collaborative nature and allows researchers and developers to easily share and reuse each other's code.

We also reviewed a number of platforms that do not apply directly to IoT but have similar general concepts that could easily be applied to it. These are Multi-Agent Systems [Fer99] platforms, which focus on orchestration of agents in distributed environments. Agents differ from the objects that we focus on, in that they take decisions based on their perceptions of the environment. Also, a fundamental conceptual distinction is that the agent's "mind" — the decision processing component — is not traditionally modifiable during run-time, as opposed to the model we are trying to achieve. The aim of Multi-Agent Systems is to create intelligent pieces of software that act on behalf of a human. Nonetheless, the methods used for messaging, distribution of tasks and rule definition, for example, provide inspiration and potentially practical support for our objectives.

Self-Managed Cells (SMC) [SBD+05] is an academic research work that proposes a policy-based federation and orchestration framework. Their architecture supports self-configuring and self-managing agents. Policies are used both for access control and for service orchestration. It is, however, a closed system in the sense that it is not build to interoperate with other systems or services. Also, a single cell implementation, albeit lightweight enough to run on mobile devices, is too heavy to run on smaller electronics[DLS+05], therefore these need to be interface through a more capable proxy device.

EVATAR [DS10] is a middleware prototype linking multiple agents with sensor, actuators and devices including robots. We are assessing EVATAR is it presents an alternative way of orchestrating physical objects — it is capable of interacting with a number of different hardware platforms by treating them as agent "bodies", while the heavy artificial intelligence processing (the "mind" of

the agent) is done on more capable computers.

Another research framework we assess is called Super-Agents [Sta10], which presents an architecture for orchestrating complex behaviours of self-governed agent organisations into a higher level abstraction called a Super-Agent. Externally, a Super-Agent acts as a single agent with policies that govern the internal behaviour. Although it is not directly applicable to our work, this framework provides interesting ideas that can be transferred to the management of groups of interacting objects.

2.5 Ad-Hoc Interactions

By definition, an ad-hoc interaction is one that is constituted or set up "in the moment", for the occasion of that concrete interaction, possibly in a scenario that was not predicted during the creation of the system that supports it. In the context of this thesis, interactions may be classified as ad-hoc on mainly two different levels.

Ad-Hoc Networks [Per08, AWD04] encompass a series of technologies and techniques that provide the means for communications between devices through a network that is created and set up for the purpose of that communication, not relying on pre-existing network infrastructures. Ad-Hoc networks are also typically decentralised.

Ad-Hoc Coordination We consider Ad-Hoc Coordination to be at a higher level than the previous kind of ad-hoc interactions. These use pre-agreed protocols to communicate information and/or coordinate actions between devices and rely on a pre-existing communication layer (not necessarily based on an ad-hoc network). Ad-Hoc Interactions are made possible through the use of standard application interfaces, shared ontologies and loosely-coupled services, which is main point of relevance to this thesis.

2.6 Summary

In this chapter, we presented the underlying concepts of the subject of our research. We began by introducing Ubiquitous Computing and its related research topics, we discuss the terminology surrounding the concept of *Smart Objects* and our initial aim of preparing the groundwork for a vision of Smart Objects orchestrated by the users using End-User Development. In order to support this

vision, we need a framework that supports EUD, ad-hoc and decentralised interactions of physical objects, whose implementation is lightweight enough to be integrated on low-powered devices, and that allows for interoperation with other systems and services. Also, the concept of Meta-Design empowers the user allowing them to further expand or modify the functionalities of objects to better suit their personal needs, making it an interesting feature to have in the framework. Not having found a framework with these characteristics, we proceed to propose our own in the next chapters.

3

Open Objects

In this chapter we explore the concept of an Open Object, what being open means in this context, what different kinds of openness it encompasses, what advantages it brings and how can we define the structure an Open Object. We then propose an architecture for a *Complete Open Object*, that is an object containing all the necessary behaviours and capabilities to operate independently in an application environment. As not all objects within an environment will be complete in terms of the behaviours and capabilities they contain, we also introduce collections of incomplete objects that can be assembled together to share their capabilities and form *Open Super-Objects* with behaviours and capabilities that are complete as a whole. To exemplify the discussion of Open Objects, we introduce throughout the chapter, a scenario that illustrates how to use our framework in a concrete application that allows people to support an activity at home.

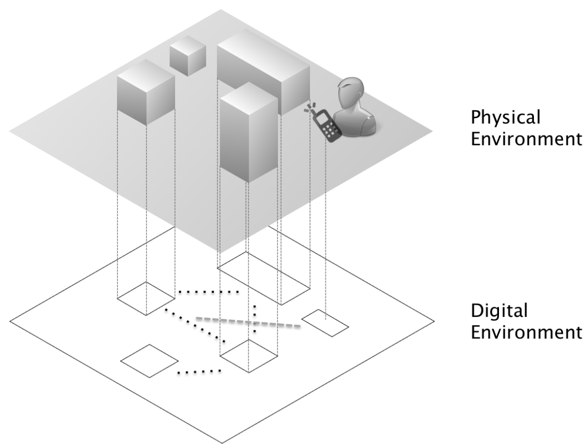

Figure 3.1: Objects in a physical environment with a representation in the digital environment. The vertical lines on the diagram illustrate the fact that the same objects have two strongly-coupled representations, one on each environment. Some entities may have only a physical (e.g. dumb objects) or a virtual (e.g. an external service) representation. A user controls a device that allows them to coordinate the interactions between the objects. Some objects may communicate through the digital environment (represented in the diagram by the dotted lines on the virtual plane)

3.1 An Environment of Open Objects

We will find Open Objects in a Physical Environment (Figure 3.1). This is a physical space that contains physical objects (for example a home or office environment). Some of the objects may contain an electronic component (or even some computational capabilities) whose software interacts with their sensors and actuators. We consider this the virtual representation of an object, as opposed to its physical representation. When this happens, we can assume that there is a digital layer underneath the physical environment, in which these physical objects are represented and through which they may communicate with each other. On the digital layer there may be some virtual objects that do not have a direct physical representation in the physical space, such as external software services. Users may interact with these objects directly on the physical layer, or through the means of interfacing objects — objects who provide a bridge interface between the two environments, such as a mobile phone or a tablet — on the digital layer. This layer of communication and computation, which we call the Digital Environment, allows for interactions between people and the objects that surround them, as well as between the objects themselves, for they can share resources and information, thus reducing redundancy, production costs and energy. It is our vision that these virtual representations of physical objects should have different degrees of openness to interact and adapt to individual people's activities.

Given this vision, we need to understand the different perspectives of openness. Openness in open-source software (OSS)[P+99], for instance, implies that developers that were not involved in the original conception of a software solution are able to have access and to modify its source code in order to create a new one that fits new needs. Similarly to OSS, we witness the uprising of Open Source Hardware (OSH) projects (notably the Arduino project[1]) where not only the firmware is made public but also the schematics for the hardware parts (i.e. mechanical drawings, schematics, bills of material, PCB layout data, HDL source code and integrated circuit layout data) so that physical artifacts can be modified, upgraded and redistributed in novel ways. Due to their open nature, OSS and OSH solutions tend to be more flexible, reliable and of higher

[1] http://www.arduino.cc

quality [R+99] than their closed counterparts.

In this work, we take inspiration from OSS and OSH, in the sense that much of the objects' functionalities could be made open and modifiable by developers, users and/or other objects. An object becomes more open when parameters that control its inner workings can be changed by the end-user, and even more so when even the inner workings can be augmented or even modified to take over new functions. The object can then be augmented dynamically with new computational capabilities that allow the object to adapt its behaviour in order to become useful in situations that were not predicted at design time. A final user being able to change an object's internal organisation and add on to how an object works, makes it "open" in terms of **Functionality**. In other words, an object's functionality becomes open-ended. This concept, allied by well-designed end-user development tools, allows objects to become much more flexible and adaptive to each user's lifestyle.

OSS and OSH are also characterised by the frequent use of open standards and protocols, which facilitate interoperability, something that is useful in our context. We refer to this kind of openness as being open in terms of **Standards**.

The idea of interoperability brings us to another view on openness: **Compositional**, when an object shares complete or partial functionalities with other objects in the environment, and it is able to make use of the capabilities shared by others. As a result, the physical and conceptual boundaries of an object become blurred: an object may not end where its functionality ends; and there may be overlaps between different object's boundaries when they start sharing software and hardware components that form their functionality. We will discuss how to compose Open Objects to share functionalities later in section 3.8.

Another perspective on openness is **Environmental**. In order for an object to be adapted to the user's lifestyle and be coherent with it, it needs to be open to the environment, be responsive to different contexts and events that happen outside its boundaries, have knowledge of other objects in the surroundings and be able to interact with them. In addition, an object may make itself noticeable in the environment, by communicating its presence and sharing information.

These four perspectives on openness present the core concepts behind our view on the fundamental element of this thesis: the Open Object. An Open Object is any physical object with a virtual representation that is capable of

observing information present in the environment (is able to receive information) in which it is situated, process it (is able to interpret, store and process data), and produce its own information for others to interact with it (is able to produce and communicate information). The Open Object paradigm is motivated by the democratisation of the object functionality creation, where the object design doesn't necessarily end on the object's designer drawing board, but can be expanded by the final user. This further development can encompass interface improvements, new functionalities, object mash-ups or connections to remote services and/or objects. In the next Chapter we will look at how one could use the framework presented here to support these interactions between objects.

We are assuming an environment of Open Objects with an underlying connectivity layer that the objects have access to and are capable of using with a common communication protocol through which objects can send messages to a list of other recipient objects. We also assume (whilst not being the aim of this study) an access control layer through which owners of the objects are able to set permissions regarding who and which other objects are able to access and modify their capabilities.

We classify physical objects as "Situated" or "Non-Situated". Situatedness has to do with how an object is related to the physical environment and context an object is. A Web service is completely non-situated meaning that it lacks a physical presence, although it may be attached to a spatial position (for instance, a weather information service). On the other hand, an object that, for example, is placed on a physical location and interacts with it is fully situated. Independently of whether an object is situated or not, it can be considered "Strongly Open", "Weakly Open" or "Closed". We consider objects to be "Strongly Open" when they allow for their functionality to be accessed and changed from the outside. On the other hand, the functionality of "Weakly Open" objects is accessible but not modifiable from the outside. "Closed" objects are objects whose functionality is neither accessible nor changeable from the outside, thus outside the scope of this thesis.

We consider that a physical object can be placed on a "smartness" scale (Figure 3.2), where on the bottom of the scale we have "dumb" objects, which do not have any representation on the virtual environment (such as a normal rock, number 1, in the picture). Number 2 on the picture shows a mug with a

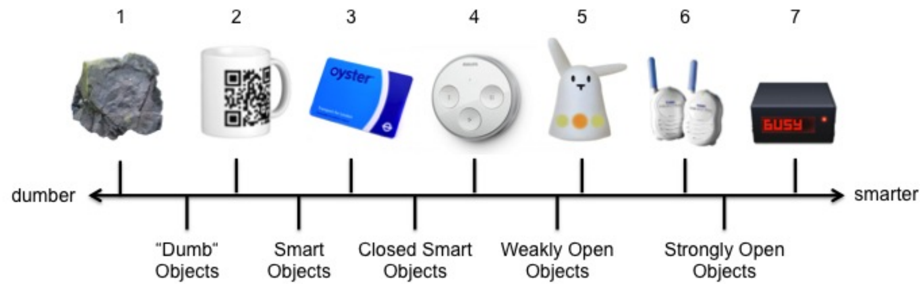

Figure 3.2: A "smartness" scale of objects.

2D barcode [KT07] attached, which allows for objects that do not have computational capabilities, to have a simple virtual representation by being able to be (uniquely or not) identifiable when scanned by an appropriate device. Next in the scale (3) we have an RFID [FW99] card, which is in fact a small, although very simple, computationally capable object, capable of simple forms of digital communication, although still being closed according to our definitions of openness above. Number 4 shows an intelligent light switch[2] that can be configured to change the colour and intensity of lights according to the user's preferences. Although not being fully "open", we consider configuration a step forward in this direction. Similarly, number 5 shows the next step towards openness, where an internet connected robotic rabbit[3] allows users to install applications and developers to create them, using their API. Number 6 and 7 are examples from our scenario (Section 3.2) and a Case Study (Chapter 6) respectively. Both are Strongly Open Objects, although the door sign (number 7) is a Hollow Object, a generic use object built without a fixed purpose and meant to be included in system developed by the user.

One of the main motivations of this thesis lies in coordination and interoperability of Open Objects and we approach both issues with the use of *Rules*. In this context, a rule is set of explicit procedures governing the activities of one or more objects. The activity can be internal, about the workings of an object, or external about the interaction of the object with other objects. These rules are typically produced and managed by users, potentially with the help of specialised software, which we call Rule Composition Tools.

[2] http : //www2.meethue.com/en − gb/the − range/hue − tap/
[3] http : //store.karotz.com/en_GB/

During the development of this work, we drew inspiration from numerous sources, namely from:

- Vlad Trifa's thesis [Tri11] — its event-drive architecture and the adoption/evaluation of the usage of REST to access device's characteristics. We build on top of these principles and propose a decentralised architecture. We didn't however adopt some components such as localisation as a component of the architecture in order to minimise its footprint and introduce the Behaviour abstraction as a means of facilitating standardisation and automation, as explained further ahead in this chapter;
- Self-Managed Cells [SBD+05], in particular Alberto Filho's thesis [SF09] — we borrowed the pattern concepts suggested on this thesis to form complex behaviour using simple objects. The design of the Open Object's rule language also borrowed ideas from this work, however, OO rules are designed with decentralisation and partial implementation in mind — the execution may begin on one device and continue on another, depending on the behaviours implemented on each device;
- The work on autonomic super-agents [Sta10] — we drew inspiration from this work for the concept of Super Open-Object, a virtual object that behaves as one to the exterior but is composed by several incomplete objects, and the rules that govern them.

We also assessed informally the interest and usefulness of our aims. This task proved to be complex as it is hard to carry on successful user surveys about a possible future scenario. In Jacob Nielson's words (based on [NL94]): "Self-reported claims are unreliable, as are user speculations about future behavior. Users do not know what they want.". We chose to take an approach similar to the one used in [CFMD10] where instead of asking subjects which objects they would combine to produce new functionalities/interactions or if they found the possibility of creating inter-object interactions, we directly present the possibility of combining the functionalities of two randomly picked existing objects. In the original experiment, the subjects were even asked to bring photos of random objects around the house, which we did not do for convenience sake. Our small informal experiment was conducted during a seminar where the subjects were

not familiar with Internet of Things and other related scenarios. We developed a computer program to pick and show two images of physical objects, common to most people's lives, at random. The subjects were then asked to propose useful interactions between the objects. It is interesting to observe that the majority of the combinations shown spawned several ideas of how to combine the functionalities of two randomly picked objects. Examples of such ideas are "Heater connected to thermometer to heat until X temperature", "Digital Picture Frame shows pictures from album X on my phone", "Oven sets its settings to the recommended when I open a specific packet", "Washing machine sends a text message to my phone when the program has ended" or "Kettle starts boiling water 1 minute before the break on my favourite show on TV". This informal survey hinted at a high degree of usefulness of a possible system that would support them and further motivated our work.

3.2 Baby Monitor Scenario

To exemplify the remaining of this chapter and the next in relation to our envisaged use of Open Objects, we introduce a simple practical scenario based around simplifying daily tasks of a family around the sleep time of their baby. This is a simplified scenario, designed to better explain the different concepts of the framework using a single example. The subject of our fictional scenario is a family of four: the two parents Charlie and Terry, one seven year old child called Mathew and one new born baby called Anna. As Anna is currently at a critical point of her development, her parents want to make sure she gets the best rest time possible. Therefore, Charlie and Terry resorted to creating a few rules for the devices around the house: the TV, the doorbell, Charlie and Terry's phones, Mathew's gaming tablet and a baby monitor. As the baby monitor is only used when the baby is sleeping, the state of the monitor (on or off) is a good indicator of whether or not it is the baby's sleeping time. The objective is to keep unnecessary noises to a minimum during this time, so the rules aim at reducing the sound volume of the devices around the house when the baby monitor is turned on:

Baby Monitor Scenario's basic rules

R1 Turn the phones in the house on silent mode;

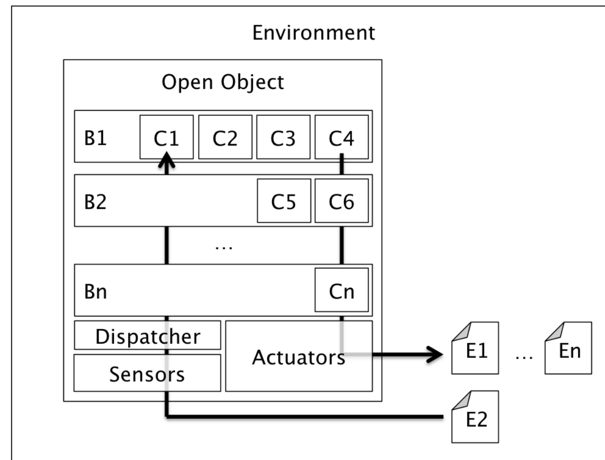

Figure 3.3: An open object consisting of a set of behaviours (B*), a set of actuators and a set of sensors. The behaviours are further defined by capabilities (C*), while the sensors and actuators receive and send events (E*) respectively, in order to interact with the environment in which the object is situated. Attached to sensors, a dispatcher forwards incoming events to the correct capabilities.

R2 The doorbell sends a text message to the phones instead of ringing;

R3 The sound volume on the TV should be limited;

R4 The sound volume on the tablet should be limited.

3.3 Open Object Architecture

To support the previously described scenario, we introduce three notions: **Capabilities (C*), Events (E*), Behaviours (B*)** (Figure 3.3) (explained next). During normal operation, an Open Object follows a Sense-Dispatch-Behave-Act cycle (Figure 3.4): An event is *received* (or *sensed*) through the sensors, it is *dispatched* into the capability it is destined to, which processes it (*behave*) and produces an output event (*act*) through the actuators.

3.3.1 Capabilities

A **Capability** is the inherent sensing, actuation or processing ability of an object to perform an action or to retrieve information. A capability can be seen as a parallel to a *method* in Object-Oriented Programming. A significant distinction between methods and capabilities is that capabilities can be added and

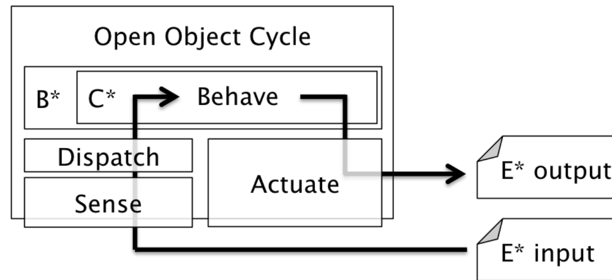

Figure 3.4: Open Objects follow a Sense-Dispatch-Behave-Act cycle during operation.

removed during runtime and they may be completely or partially implemented on an object other than the one they belong to conceptually; in other words, an object's capability may be stored and executed on another objects (or even group of objects) in order to conserve power on the first object, for instance.

An Open Object is one that is able to share its capabilities with other objects but also to outsource capabilities when needed. A capability may be considered *atomic* (lower-level in nature) or *complex* (composed from atomic capabilities, coordinated at a higher level). Capabilities can also be classified according to their origin. In this sense they can be *innate*, when they were hard-coded and embedded into the object at design time, or *acquired*, when during the life-time of the object a new functionality was added to it. The mechanisms that allow a capability to be acquired are, themselves, capabilities and are described in detail in the next chapter. New functionalities can be added by the user (or another object) when coordinating the object's existing capabilities to form a new, complex one.

In our scenario, the objects have the following capabilities:

TV
C1 Set Sound Volume
C2 Set Max Sound Limit
C3 Change Channel
C4 Switch On
C5 Switch Off

Phone
C6 Set To Silent Mode
C7 Set To Normal Mode
C8 Send Text Message

Doorbell
C9 Ring

Tablet
C10 Set Sound Volume

Baby Monitor
C11 Switch On
C12 Switch Off

3.3.2 Events

An **Event** is a description of a happening (along-side with other meaningful information such as a timeframe and origin of the event) and it is said to belong to an Event Type, which classifies the happening and allows objects to respond to them in standard ways.

Although, in practice, all events are treated the same way throughout the system, there are two types of events that are conceptually different as they have slightly different purposes: *Capability Requests* and *Inform Events*. Event requests are typically produced as part of a rule and the objective, as the name implies, is to request an action or a value from another behaviour/object. When an object produces an Inform event, on the other hand, it does not have any particular intention of producing an action or requesting values from other objects, it merely serves the purpose of informing the objects in the environment of an internal or external state change; other objects may or may not have rules that are triggered by this event but they are independent from it and, as a result, are part of a different "conversation" — a sequence of related events which are responses to each other.

For example, in our scenario, the doorbell is able to produce an event "doorbell rang" event, when someone presses its button, and the baby monitor is able to produce the events "noise detected", "switched on" and "switched off".

Events are used as messages between (and within) objects when requesting Capabilities. In fact, all communication between capabilities is done through the exchange of events. These events may also trigger the execution of certain rules, for example, in our scenario, the rules R1, R2, R3 and R4 are triggered when the baby monitor produces a "switched on" event.

3.3.3 Behaviours

Objects' capabilities are grouped together to form *behaviours* that the object can display in interactions with other objects in order to serve a certain *purpose*. Behaviours and purposes facilitate interactions by describing and generalising common procedures and produce an expectation on the object. A behaviour can be seen as a parallel to interfaces in Object-Oriented Programming. The behaviour of an Open Object for a given purpose is the aggregation of all the capabilities and events that the object exposes to the environment to serve that

purpose. As an open object can serve different purposes, it can exhibit different behaviours. By sharing behaviours, they also allow each object to focus on fewer tasks, and interact more effectively. Objects can, therefore, *display* a set of behaviours to the outside environment. In practice, this allows simpler, lighter-weight objects to make use of more complex behaviours that others, more capable, objects display and make better use of their capabilities and avoid redundancies.

We classify behaviours into three categories:

Domain Dependent Behaviours are behaviours whose domain the user wants to control directly, as their primary aim (e.g. the user's television or the baby monitor).

Domain Independent Behaviours offer generic support that the user may use to support and reuse across his applications (e.g. mathematical, application flow control or memory storage related behaviours). These differ from the domain dependent behaviours in the sense that they merely provide tools that assist the user in reaching its goal. For example, if a user wants to limit the volume of the television after a certain time of the day, Television would be a domain dependent behaviour, and Time Keeping would be a domain independent one.

System Behaviours are behaviours that manage the system itself and include tasks such as coordination and service registration. These are explained in greater detail in the next section.

Behaviours are one way of maintaining the lightweightness of the framework: every system component is isolated and works independently as a distributable system behaviour. System Behaviours form the bare minimum core set of functionalities. All accessory functionalities (not fundamental to the basic structure of the system) are encapsulated in domain independent behaviours. Any of these behaviours can be displayed by more than one object. System behaviours, for example, can be displayed by several different objects in order to maintain resilience of the system when some objects become unavailable.

In our scenario we have different objects displaying behaviours of different kinds. A few examples of domain independent and domain dependent behaviours follows. Objects may display secondary behaviours, which are not related to its main function.

TV	**Phone**	**Tablet**
B1 Television	B7 Telephone	B13 Time Keeping
B2 Generic Appliance	B8 Time Keeping	B14 Message Displayer
B3 Radio	B9 GPS	B15 Web Client
B4 Time Keeping	B10 Message Displayer	
B5 Message Displayer	B11 Web Client	

Doorbell	**Baby Monitor**
B6 Doorbell	B12 Baby Monitor

3.4 System Behaviours

System Behaviours are behaviours that manage the operations and interactions within an Open Object environment. There are five System Behaviours in the Open Object framework: *Behaviour Mapping (BM)*, *Rule Execution (RE)*, *Rule Repository (RR)*, *Event Broker (EB)* and *Behaviour Advertising (BA)*. In order for a group of Open Objects to operate as intended, all five System Behaviours need to be reachable by either being present in the environment or accessible from a remote location. We describe the function of each of these next.

3.4.1 Behaviour Mapping

The Behaviour Mapping maps registered behaviours to the objects that display them. Its purpose is to provide information about which objects are able to perform which tasks when the respective behaviours are required. It also performs an active behaviour discovery process, where it looks for and registers new objects and their behaviours in the environment. Through either method, an object that is registered in the behaviour mapping behaviour is also able to query it regarding other behaviours, hence this behaviour serves as a gate to the environment: when one object's behaviour needs to request a capability from another behaviour, it can query the Behaviour Mapping in order to retrieve the object's address which displays that behaviour.

A previously defined map of behaviours-objects may also be provided to the Behaviour Mapping for less ad-hoc situations, when, for example, the user wants to specify exactly which objects perform which tasks.

3.4.2 Rule Execution

As the name implies, this behaviour is able to interpret rules and request the right capabilities as defined per rule definition.

3.4.3 Rule Repository

The Rule Repository behaviour stores and provides rules when requested.

3.4.4 Event Broker

The Event Broker has two main tasks: maintain a registry of subscriptions for events — which object is subscribing to which types of events — and serves as an intermediary between senders and recipients of events according to the subscriptions.

3.4.5 Behaviour Advertising

This behaviour advertises which behaviours objects display. Typically this behaviour is displayed by each object and advertises its own behaviours, but it may also provide information about third-party objects and external services. This behaviour may also actively register behaviours in a Behaviour Mapping behaviour.

System Behaviours in the Scenario

In our fictional scenario some or all of the objects involved may also display system behaviours on top of the domain specific ones referred before. If not all five system behaviours are displayed by these objects, the system would need to request the use of additional objects to fulfil this task (a desktop computer, or an external server, for example). For the sake of simplicity we're considering that the system behaviours are displayed and distributed as follow:

- **TV** Rule Execution; Rule Repository; Behaviour Mapping; Behaviour Advertising
- **Baby Monitor** Event Broker; Behaviour Advertising
- **Phone / Tablet / Doorbell** Behaviour Advertising

The reason why the TV is displaying more system behaviours than other objects is that even if others also display some or all of these, the TV would

be the preferred object due to having more extended memory/computational capabilities and not relying on batteries. It is not compulsory that every object displays behaviour advertising but it is a likely scenario, as it makes sense for each object to advertise their own behaviours directly, considering that system behaviour are the most lightweight in terms of hardware/software requirements. Another alternative possibility would be to distribute the system behaviours as follows:

- **Baby Monitor** Rule Execution; Rule Repository; Behaviour Mapping; Event Broker; Behaviour Advertising
- **Phone / Tablet / Doorbell / TV** Behaviour Advertising

This alternative distribution is heavier on the Baby Monitor but it has interesting possibilities: it allows the user to bring along the system embedded in the baby monitor if they go to a friend's house or to a hotel, so that the same rules will apply in other spaces (assuming that the objects in the new spaces are also Open Objects). We could also have hybrid systems with redundancy in terms of system behaviours and rules, in order to support either scenario.

3.5 An overview of the framework

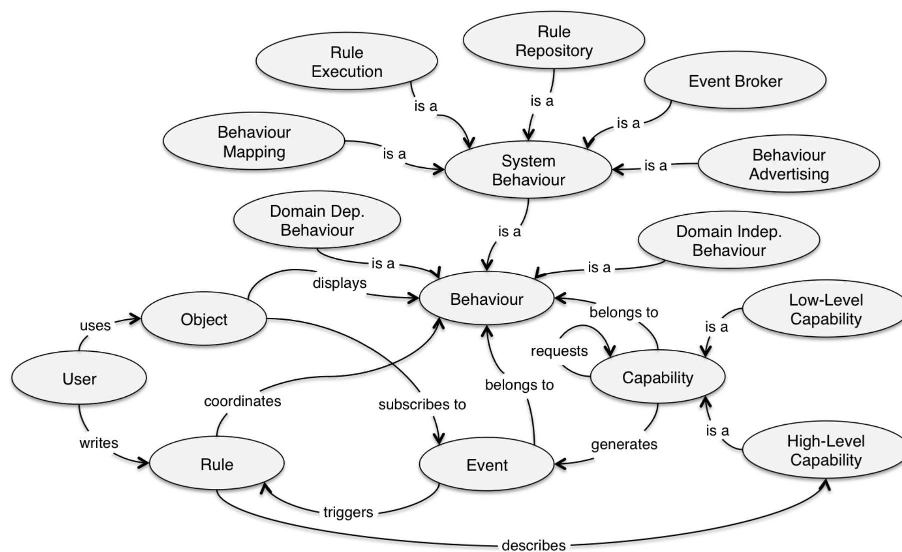

Figure 3.5: The global Open Object Ontology.

The concepts discussed so far form the Open Object ontology (Figure 3.5) which provides a "big picture" perspective over the framework and shows interactions between the different components. With this picture in mind, we consider that the scope of the Open Objects (Chapter 3.6) places the framework between a physical environment which is accessed through the use of sensors and actuators and the user's activities, which in this context are supported by rules (governed by the system behaviours). Low-level capabilities bridge object's hardware and the framework by exposing functionality that uses hardware API's to interact with the sensors and actuators.

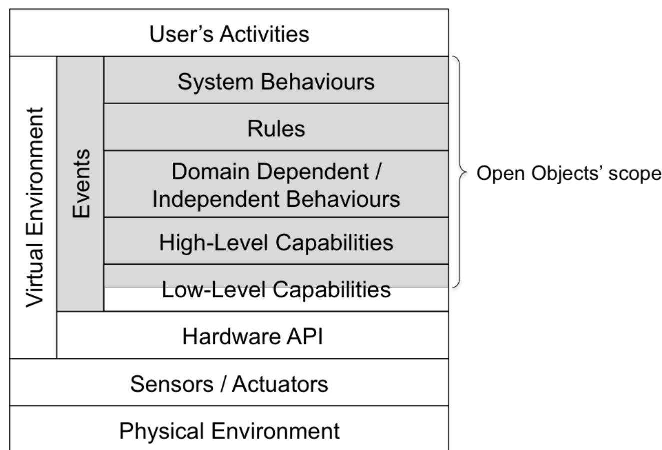

Figure 3.6: The scope of the Open Objects framework.

In the next section, we take a closer look over how System Behaviours interact to support Open Object Systems.

3.6 Interactions between System Behaviours

We proceed with discussing the processes and interactions between system behaviours understood here as the implicit rules that govern the overall system behaviour. The diagrams in this section offer a high-level view of these interactions where arrows show the main flux of information. On certain diagrams, "Domain Dependent / Independent Behaviours" represent any behaviour present in the system (including system behaviours although each of them is represented individually).

3.6.1 Event-based Interaction

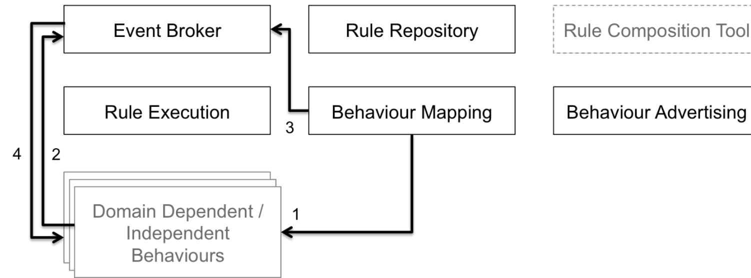

Figure 3.7: The event passing process.

As mentioned previously, all messages between behaviours, including interaction between system behaviours, are described via events. Therefore, all interactions described further in this section are based on the one described below (and in Figure 3.7) in order to support interactions between behaviours.

When an event is produced by a behaviour, this behaviour asks a behaviour mapping behaviours to which it is registered to about which objects display the Event Broker behaviour (step 1 in Figure 3.7). It then sends the event to the event brokers (step 2). The event brokers then asks the behaviour mapping behaviours about which objects display the recipients' behaviours (step 3) and also checks if there are any subscriptions for this event. It then forwards the event to the recipients (the objects returned by the behaviour mapping behaviours) and to the subscribers (step 4). In the scenario, for example, the "baby monitor" behaviour produces a "switched on" event and sends it to the Event Broker behaviour on the baby monitor. The Rule Repository (on the TV) had subscribed to the baby monitor's "switched on" event, as there is a rule that is triggered by it, therefore it receives it as well, triggering the execution of the rule.

Steps when behaviours query the Behaviour Mapping (steps 1 and 3) for objects may be skipped if there had been a recent (period of time to be defined per application) query to minimise communication overhead, as in [MD88]. In a case where these steps are skipped but the recipient behaviour is not available anymore, these queries would then be performed.

3.6.2 Behaviour Registration

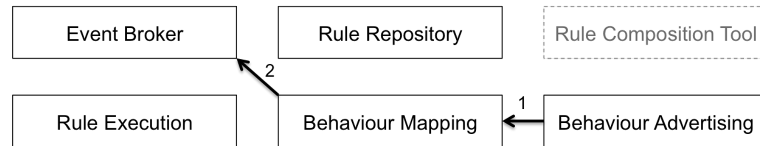

Figure 3.8: Behaviours are registered in the Behaviour Mapping behaviour by the Behaviour Advertising behaviour.

A Behaviour Advertising behaviour registers a list of behaviours (step 1 on Figure 3.8) that are displayed by a given object. During the registration process, the Behaviour Mapping behaviour may also optionally automatically create a subscription for events related to these behaviours, to the respective object (step 2). When the system starts, for example, the objects in the environment register their behaviours in the Behaviour Mapping on the TV (including the other behaviours displayed by the TV), so that they are accessible during the lifetime of the system.

3.6.3 Behaviour Discovery

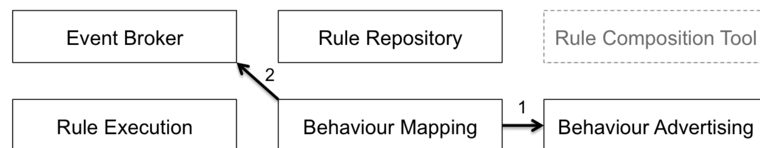

Figure 3.9: behaviours are discovered and registered by the Behaviour Mapping behaviour.

During the discovery process, a Behaviour Advertising behaviour registers a list of behaviours that are displayed by a given object (step 1 on Figure 3.9) and, as described on the previous process, subscribes to the related events (step 2). Objects are unregistered if they are not accessible after a period of time (defined per application).

For example, if a new phone enters the scenario environment, it is discovered by the Behaviour Mapping behaviour on the TV, registered, and requested to lower its sound volume, when the rule dictates so.

3.6.4 Rule Composition

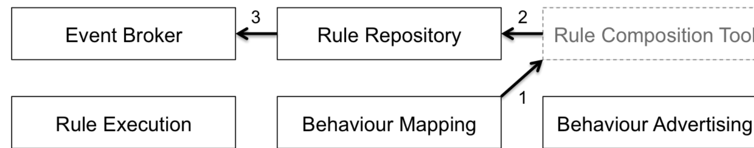

Figure 3.10: A user, through a Rule Composition Tool, creates a rule.

A Rule Composition Tool requests from Behaviour Mapping a list of the behaviours available in the environment (step 1 on Figure 3.10) and, based on these (but also potentially other behaviours manually specified by the user), assists the user through the creation of the rule. When the user is done with this task, the rule is then stored in the Rule Repository (step 2). When registering a rule, the Rule Repository also subscribes to the event to which the rule applies (step 3).

In the scenario, the user uses a Rule Composition Tool app on their phone to create the rule. The app receives a list of the behaviours which are available in the environment, from the Behaviour Mapping behaviour on the TV and stores the rule on the Rule Repository behaviour, also displayed by the TV.

3.6.5 Rule Execution

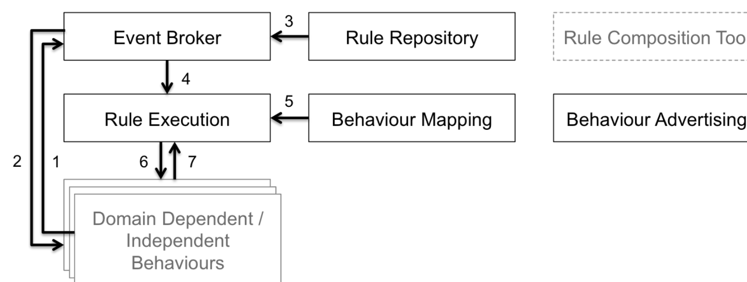

Figure 3.11: Events are produced and forwarded to the correct recipients. Rules may also be triggered by the events.

When an event is produced (typically by a Domain Dependent or Domain Independent Behaviour — step 1 on Figure 3.11) it is sent to an event broker, which checks the subscriptions for that kind of event. If one or more domain

dependent or domain independent behaviour had subscribed to that event type, the event is forwarded their way (step 2). If there is a rule that responds to this event type (step 3), the rule is forwarded to rule execution behaviour (step 4). During the execution of the rule, a behaviour mapping behaviour is requested to query about which objects should be used at each step of the execution (in other words, which objects map to each behaviour present in the rule — step 5). Next, the respective capabilities on the correct object's behaviours are requested (step 6). These capabilities may request the execution of certain parts of the rule or may request the usage of certain capabilities (a capability request is considered an atomic component of a rule, and is also interpreted by the Rule Execution — step 7), depending on their internal logic.

In the scenario, when the baby monitor produces a "switched on" event, it is sent to the Event Broker behaviour (also displayed by the baby monitor object) for broking. As there is a rule that is triggered by this event, the rule is sent to the Rule Execution behaviour (displayed by the TV) for parsing and execution. During execution, the rule may dictate the request of certain capabilities from each object in the environment: for instance, the rule dictates that the Phone should be set to silent, therefore the Rule Execution behaviour "asks" the Behaviour Mapping which object is displaying the "Phone" behaviour which will return the addressable phone object; once it knows this information, the Rule Execution is able to request the "Set To Silent" capability of the Phone behaviour, displayed by the phone object.

3.7 A Complete Open Object

We call Complete Open Object (Figure 3.12) an object that is able to function independently, while still displaying the three kinds of openness described in section 3.1 before. It is complete due to the fact that, in order to function independently, it displays (and implements) all five System Behaviours. In that sense, this kind of object is ideal but unrealistic in many cases considering that the necessary computing power, memory and power consumption, although trivial for a common personal device, may be too much for electronic chips similar to those found in RFID tags, for example. Also, it can be considered redundant and not optimal to have all System Behaviours on all objects, when they could be shared. Due to these assumptions, sharing resources was one of

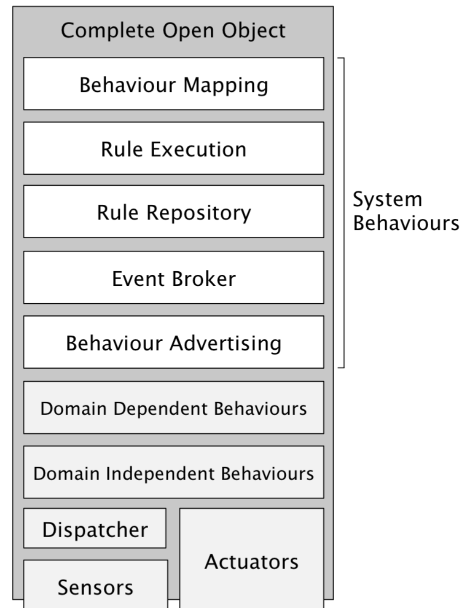

Figure 3.12: A complete object, with the five System Behaviours plus Domain Dependent and Domain Independent Behaviours

the main concerns when designing the framework. Objects may not only share their domain independent and domain dependent behaviours, but also their system behaviours, as described in the next section.

3.8 Open Super-Objects

Objects that do not display all System Behaviours or require other objects' behaviours in order to fulfil a certain task need to collaborate with objects that display said behaviours (Figure 3.13). In order for a collaboration to be possible, all the domain dependent and domain independent behaviours required for the collaboration and the five system behaviours need to be displayed by the objects participating in the collaboration. When this happens, we say we are in the presence of a Open Super-Object (a concept borrowed from the multi-agent systems world [Sta10]): a group of Open Objects that act independently as a single one, with Rules that govern itself internally and its external behaviour. We can think of these rules as the "glue" that connects and regulates the atomic objects within an Open Super-Object (Figure 3.14). In the scenario, all the objects work together as a single entity, regulated by the rules specified by the

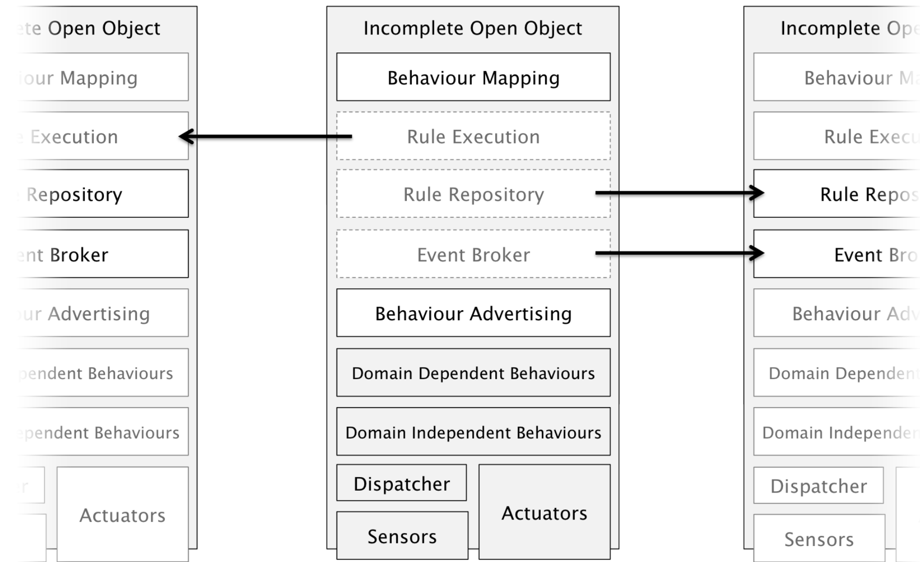

Figure 3.13: Incomplete object sharing System Behaviours

user, towards a common goal: to facilitate and improve the baby's sleeping time. Therefore we can say that we are in the presence of an Open Super-Object.

3.9 Summary

In this Chapter we broke the definition of an Open Object down to its concepts and components. We discussed different kinds of openness in this context, and how they apply to the Open Object framework and explored the three major building blocks of the Open Object environment: Events, Behaviours and Capabilities. We also observed how System Behaviours work together to support an Open Object environment and presented the concept of Open Super-Object, a single independent entity composed of several objects that work towards a common goal, with rules that regulate its internal behaviour. In the next Chapter, we propose a formal specification for each of these components.

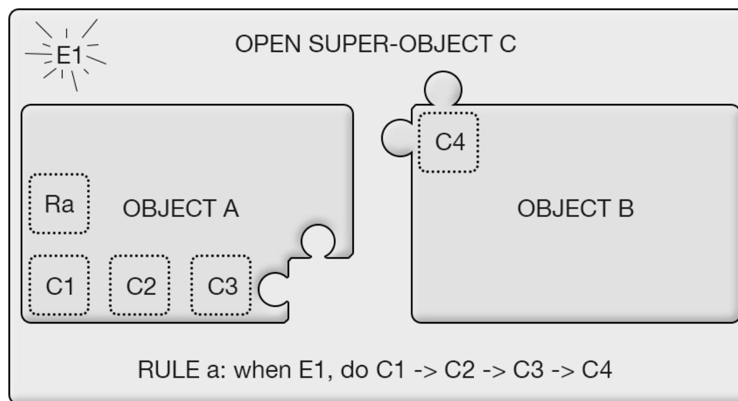

Figure 3.14: Two objects collaborate according to a rule, forming an Open Super-Object. In the diagram Rx represent rules; Ex represent events and Cx represent capabilities. Ra/Rule a is defined by the sequential execution of C1, C2, C3, C4 given event E1.

4

Open Object Description Language: Specification and Implementation

In this chapter we propose a systematic specification of the Open Object Description Language (OODL) of each component and data format of the Open Object framework, discussed in the previous chapter. This specification provides a generic, technology independent platform onto which to build an implementation, such as the one in the next chapter. In order to specify the organisation of each component and data format, we use a variant of Backus-Naur Form. Alongside, we also illustrate each specification with a practical example from the scenario using the implementation syntax.

4.1 Our Specification Approach

We aim at creating a framework that is able to support interactions of the kind shown by the scenario in Chapter 3 on low-powered and computationally

lightweight devices. Therefore, we intend to define a scalable, vertically decentralisable system able to work according to these limitations. By vertically decentralisable we mean a system that is modular from the application layer to the core system management functionalities. To pursue this aim we designed a modular system of atomic behaviours that can be individually distributed and a series of lightweight and cross-platform description format specifications.

In order to represent these specifications, we choose a variant of the Backus-Naur Form [Gar05] (BNF) as a structure definition language. BNF is a language that is usually used to specify programming languages, but it is also effective as a small, easily readable format for specifying data structure and component organisation textually. BNF usually has a top-down approach starting with a root and further specifying each term as it appears in the definition. A textual example: “A *Vehicle* can be *Aerial*, *Terrestrial* or *Aquatic*; a *Aerial Vehicle* can be a *Airplane* or a *Helicopter*; a *Terrestrial Vehicle* can be a *Car* or a *Bicycle*; a *Car* has 3 or more wheels; a *Bicycle* has 2 wheels; ...”.

BNF assists us with defining how each component is organised and what is the relation between components. In our BNF variant, we use the set of symbols in Listing 4.1. Some of the specifications ahead are not self-contained, as some of the terms are defined on specifications described further ahead.

Listing 4.1: Symbols used in our BNF Variant.

```
::=      — Defined as
(space) — Concatenation
<...>    — Non Terminal, to be defined
?        — Optional, zero or one of the following component
*        — Zero or more of the following component
+        — One or more of the following component
|        — Alternation, The previous component Or the next
TEXT     — Terminal, text in capitals indicates a term that
    is not defined further
TYPE     — A terminal value that belongs to the data type
    indicated
```

In parallel to the specification, we exemplify each case with a translation from the generic specification into the formats used in our implementation. Although we could have also used a proprietary compressed binary format —

particularly for rules and events, where performance is more critical — which would arguably be a better solution for real-world applications, we decided to use JSON[1] for all the data formats in the framework. The main reason is that JSON offers us a lightweight, cross-platform and web-friendly [Wan11] language which makes it easy and fast to perform tests across different platforms and to use different programming languages. JSON's organisation lies in a structure of labeled values, objects and arrays, using the following set of structural elements (Listing 4.2).

Listing 4.2: Symbols used in our JSON implementation.

```
"KEY": VALUE - Defines a pair of key-value. Keys are always
    strings.
VALUE        - A value can be a string, a number, "true", "
    false", "null", an object or an array.
{...}        - An object is a series of key-value pairs
    separated by commas (,) and delimited by curly brackets.
[...]        - An array is a collection of values separated
    by commas (,) and delimited by square brackets.
```

4.2 Open Object

An Open Object is defined in Listing 4.3 below. Although this definition helps understand the rest of the chapter, it is merely conceptual and does not have an explicit representation in the system.

Listing 4.3: Open Object specification.

```
<Open Object> ::= +<Behaviour> *ACTUATOR *SENSOR DISPATCHER
<Behaviour> ::= *<Capability> *<Event> STATE
<Capability> ::= <Low-Level Capability> |
    <High-Level Capability>
<Low-Level Capability> ::= <Capability Description>
    MACHINE-CODE
<High-Level Capability> ::= <Capability Description> <Rule>
```

As shown by the above specification, an Open Object has actuators, sensors,

[1] "http://json.org/"

a dispatcher and displays a set of behaviours. A behaviour has capabilities, events and an internal state. Capabilities can be either low-level (hard-coded in the object) or high-level (described by a rule).

4.2.1 Open Super-Object

An Open Super-Object is a group of objects that, between them, display all system behaviours, making it capable of working independently. An Open Super-Object behaves as a single object and is governed by a set of rules. A complete Open Object is a specific case of an Open Super-Object, with only one object. Alike the previous definition, the Open Super-Object specification is merely conceptual and implicit in the framework.

From a behavioural point of view, we define an Open Super-Object in Listing 4.4 below.

Listing 4.4: Open Super-Object specification.

```
<Open Super-Object> ::= +<Open Object> <System Behaviours>
    +<Domain Dependent Behaviour> +<Domain Independent
    Behaviour> +<Rule>
<System Behaviours> ::= <Behaviour Mapping> <Rule Execution>
     <Rule Repository> <Event Broker>
<Domain Independent Behaviour> ::= <Behaviour>
<Domain Dependent Behaviour> ::= <Behaviour>
<Behaviour Advertising> ::= <Behaviour>
<Behaviour Mapping> ::= <Behaviour>
<Rule Execution> ::= <Behaviour>
<Rule Repository> ::= <Behaviour>
<Event Broker> ::= <Behaviour>
```

4.3 Capabilities

The Capability Description (defined in Listing 4.5), defines and informs how to use a capability. It is the equivalent to a method signature in object-oriented programming. Below is the specification of the Capability Description in the BNF variant.

Listing 4.5: Capability Description specification.

```
<Capability Description> ::= <Capability Name> ?<Description
    > <Output Type> *<Required Behaviour> ?<Execution Mode>
    *<Parameter Definition> ?<External>
<Capability Name> ::= STRING
<Description> ::= STRING
<Output Type> ::= DATATYPE | NONE
<Required Behaviour> ::= STRING
<Execution Mode> ::= Top-down | Bottom-up
<Parameter Definition> ::= <Parameter Name> <Type> ?<
    Required>
<Parameter Name> ::= STRING
<Required> ::= TRUE | FALSE
<Type> ::= STRING
<External> ::= STRING
```

This specification tells us that a capability description is identified by a string identifier (name), may contain a human readable description and zero or more parameters. A parameter is defined by a name, a type and a flag that states if this parameter is required or if, otherwise, it can be omitted. Execution mode is explained in Section 5.6 and the default value is *Top-down*. In the case of an external capability (a web service, for example), the "External" parameter defines the source. On a web environment, for instance, it states the URL of the web service.

Below (in Listing 4.6), we demonstrate how the previous specification translates into our JSON implementation. We use an example from the scenario, the TV behaviour's "SetSoundVolume" Capability Description. In the example, a Capability called"SetSoundVolume" does not have an output, requires the "Math" Behaviour and takes one required numeric parameter "volume".

Listing 4.6: Capability Description example.

```
{
  "capability": "SetSoundVolume",
  "description": "Sets the volume according to the
     percentual value provided",
  "output": null,
  "required": ["Math"],
  "parameters": [
    {
      "name": "volume",
      "type": "NUMBER",
      "required": true,
    }
  ]
}
```

In order to use external capabilities (such as Web Services), from within an Open Object environment, we need to define the "External" parameter of the definition. In order to pass parameters to a Web Service, we use the $PARAMETER syntax, as shown in Listing 4.7.

As previously explained, all communication between capabilities is done by exchanging events, therefore capability requests are performed by events, specified in Listing 4.9, in the next section.

Listing 4.7: External Capability Description example.

```
{
  "capability": "CurrentWeather",
  "description": "Provides the current weather information
      for a given location",
  "external": "http://api.openweathermap.org/data/2.5/
      weather
      ?q={$location}",
  "output": "Json Object",
  "parameters": [
    {
      "name": "location",
      "type": "STRING",
      "required": true,
    }
  ]
}
```

4.4 Events

Below, in Listings 4.9 and 4.8, are the definition of an event type and the specification of an event itself, respectively. While the former defines a generic event "skeleton" to which events comply, the latter defined the format used for transmitting the events themselves.

Listing 4.8: Event Type Definition.

```
<Event Definition> ::= <Behaviour Name> <Event Name> *<
    Property>
<Behaviour Name> ::= STRING
<Event Name> ::= STRING
<Property> ::= <Property Name> <Property Type>
<Property Name> ::= STRING
<Property Type> ::= TYPE
```

When specifying an Event Type one defined which behaviour the event is originated from, the name of the event and it's properties.

Listing 4.9: Event specification.

```
<Event> ::= <Behaviour Name> <Event Name> ?<Id> ?<From> *<To
    > ?<Conversation Id> ?<Visibility> ?<In Response To> ?<
    Execution Mode> ?<On Error>  *<Property>
<Behaviour Name> ::= STRING
<Event Name> ::= STRING
<Id> ::= STRING
<From> ::= STRING
<To> ::= STRING
<Conversation Id> ::= STRING
<Visibility> ::= Public | Private
<In Response To> ::= STRING
<Execution Mode> ::= Top-down | Bottom-up
<On Error> ::= +<Event>
<Property> ::= <Property Name> <Property Value>
<Property Name> ::= STRING
<Property Value> ::= STRING | +<Event>
```

What the previous specification tells us is that an event is characterised by the behaviour in which it is originated, its event name and properties. Properties have a name and a value. Each component is explained below.

Event Id uniquely identifies the event so that it can be referenced to later in the conversation, in the "in response to" field, for example.

To specifies the recipient of the event (typically a URI). The recipient is not compulsory as it may be provided by the Event Mapping (Chapter 5).

Conversation Id uniquely identifies the conversation. A conversation is a sequence of events that are produced and shared in response of a another previous event, just like a human conversation is a series of interactions between a set of people in response to one another.

Visibility indicates if an event is Public: every object is able to "see" an event whether or not it is in the recipient list; or Private: only objects in the recipient list receive the event.

In Response To defines which event id's it is a response to, in case the event is a response to a Request.

Properties - An Event may contain certain context-specific properties that

describe the happening or configure the request. In the case of an request event, typically events produced during the execution of a rule, properties can be linked to the output of other events, creating a tree of event interactions. Execution Mode describes the sequence in which these events are requested, and is explained in Section 5.6 in the Rule Execution Behaviour specification. By linking property values to the outcome of other capabilities request, we are in fact creating a data-flow language [Hil92]. This is done with the intention of facilitating the mapping between rules and a data-flow centric End-User Development tool. Properties can be used in rules by referencing them either in the rule condition or as property values of the actions (Section 4.6).

In Listing 4.11, an Inform Event, defined in Listing 4.10 is raised as a sound is detected by the baby monitor. The event contains a property "volume" which provides information about the intensity of the sound detected.

Due to the different nature of Inform and Capability Request Events, properties are also called parameters in the case of the latter.

Listing 4.10: A sound detected event definition example.

```
{
  "event": "BabyMonitor.SoundDetected",
  "properties": [
    {
      "name": "volume",
      "type": "Number"
    }
  ]
}
```

Listing 4.11: A sound detected event example.

```
{
  "event": "BabyMonitor.SoundDetected",
  "id": "e1",
  "from": "babymonitorA",
  "to": "tvA",
  "cid": "cid0",
  "visibility": "public",
  "volume": 5
}
```

Below, in Listing 5.7, a capability request event shows a request of the ”Set Sound Volume” capability (in an implementation independent syntax, merely as an example), in order to set the volume to 25.

Listing 4.12: Capability Request event example.

```
{
  "event": "Television.SetSoundVolume",
  "id": "e2",
  "from": "babymonitorA",
  "to": "tvA",
  "cid": "cid1",
  "visibility": "private",
  "volume": 25
}
```

In the case of an external capability request, we state the external URI in the “to” parameter, as can be seen in Listing 4.13.

Listing 4.13: External Capability Request event example.

```
{
  "event": "Weather.CurrentWeather",
  "to": "http://api.openweathermap.org/data/2.5/weather?q={$
      location}",
  "location": "London,uk"
}
```

As previously mentioned, requests can be chained together by specifying further requests for the “On Error” field or for Property fields. In Listing 4.14 we show an example of request chaining. In this example we use two capabilities — *Sum* and *Multiply* — of the *Math* behaviour. Both accept A and B properties to which we provide values, except for the property B of the Sum capability, which we chain to the response of the Multiply capability request. In this case, Multiply responds with “10” and Sum can then calculate its own response of “12”.

Listing 4.14: Capability Request chaining example.

```
{
  "event": "Math.Sum",
  "A": 2,
  "B": {
    "event": "Math.Multiply",
    "A": 2,
    "B": 5
  }
}
```

An event belongs to an event type which classifies it and makes its usage more predictable. Event Types are composed of two parts separated by a dot (”.”): BEHAVIOURID.EVENTID (notation that can be observed in the examples above). This makes it easier to uniquely identify and select an event. We can used wildcard event selectors such as:

- doorbell.rang - selects all “rang” events from the “doorbell” behaviour;
- doorbell.* - selects all events from the “doorbell” behaviour;
- *.rang - selects all “rang” events from any behaviour;
- *.* - selects all events;

Event selectors are especially useful for event subscription as will be explained ahead, in Section 5.4.

4.5 Behaviours

A behaviour is characterised by a *Behaviour Description* which objects advertise and to which they comply when implementing the behaviours components. As an object may exhibit several different behaviours, it may comply with several different behaviour descriptions. A behaviour description is used when composing rules and is partly user-centric, to let the user understand what to expect from and how to use the object. The behaviour description is what the object advertises when it wants to let other objects know which behaviours it displays and what it provides when others inquire for it.

Listing 4.15: Behaviour Description specification.

```
BEHAVIOUR_DESCRIPTION ::= <Name> ?<Description>
    *<Capability Definition> *<Event Definition>
    *<Characteristic Definition>
<Name> ::= STRING
<Description> ::= STRING
<Characteristic Definition> ::= <Parameter Definition>
    ?<Description>
```

In the previous specification, characteristics describe properties of the objects that are useful when choosing between different objects that display the same behaviour. For example, in Listing 4.16, a *Television* behaviour has a *Resolution* characteristic that can be used if a user defines a rule to show a certain video in the highest resolution screen in the environment. Television Behaviour Description describes how to interact with a Television object through its Television Behaviour (The object may also exhibit other behaviours, such as Time Keeping or Message Displayer) by describing its capabilities — how to request information or execution — and its events — how to trigger the execution of rules following internal happenings or state changes in this object.

Listing 4.16: Behaviour Description example.

```
{
  "behaviour": "Television",
  "description": "A generic television",
  "capabilities": [
    {
      "capability": "Television.SetSoundVolume",
      "description": "Sets the volume according to the
          percentual value provided",
      "output": null,
      "required": ["Math"],
      "parameters": [
        {"name": "volume",
         "type": "NUMBER",
         "required": true}
      ]
    }
  ],
  "events": [
    {
      "event": "Television.TurnedOn"
    },
    {
      "event": "Television.VolumeChanged",
      "properties": [
        {"name": "volume",
         "type": "Number"}
      ]
    }
  ],
  "characteristics": [
    {"name": "Resolution",
     "required: false,
     "type": "Number"}
  ]
}
```

4.6 Rules

Rules are a sequence of capability requests that are triggered by one or more event types. This sequence of capability requests are themselves events, that may be connected to further capability requests, as shown in Listing 4.14. We are using a specific kind of rules which falls under the Event-Conditon-Action class of rules (ECA rules [PPW06]) whose general syntax is "on *event* if *condition* do *actions*" These rules are triggered by an event, and if the conditions apply, the actions are performed. Conditions can compare event parameter against values or against other parameters. Our rules specification follows, in Listing 4.17. Event parameters can also be used as parameter values in the actions (capability requests specified in the "action" field of the rule).

Listing 4.17: Rules Specification.

```
<Rule> ::= <Identifier> +<Event Name> <Conditions> <Actions>
<Event Name> ::= STRING
<Identifier> ::= STRING
<Conditions> ::= <Parameter Name> <Comparison Operator>
    <Value> | <Parameter Name> *(<Logical Operator>
    <Conditions>)
<Comparison Operator> ::= < | <= | == | != | => | >
<Logical Operator> ::= AND | OR | NOT
<Value> ::= STRING
<Actions> ::= +<Event Description>
```

Listing 4.18: Rule example.

```
{
  "rule": "R3",
  "event": "BabyMonitor.soundDetected",
  "conditions": "volume > 5",
  "actions": [
    {
      "event": "Television.SetSoundVolume",
      "volume": 25
    }
  ]
}
```

in Listing 4.18 we show an example rule from the scenario, where a sound detected event triggers rule R3 into requesting the television to set the sound volume to 25, if the detected sound volume exceeds 5.

4.7 Summary

Based on the concepts presented in Chapter 3, in this chapter we proposed a generic and systematic specification of each component in the framework and exemplified it with specific cases from the scenario, translated into the implementation language. We used a variant of BNF to explain the internal organization of each component of the framework as well as to specify the structure of different formats of the OODL. We also explained here how we translated these into our implementation formats, for which we used JSON. In the next Chapter we define each System Behaviour along with their events and capabilities, and some implementation strategies and issues.

5

Implementation

In Chapter 4 we theorised the generic and distributed Open Objects framework. Having specified each component of the framework in general terms, in this chapter we proceed to specifying the internal functionality of each system behaviour. For each of these, we define an internal state, the associated events, and the object-oriented pseudo-code procedure for the non trivial capabilities. The pseudo-code for all system behaviours' capabilities can be found in Appendix A. We also discuss (Section 5.7) our web oriented approach to the implementation of the framework in three different platforms. For this implementation we chose to use Web technologies [Cos07, BLFF96] to support communication (following the direction of the recent trends in Smart Objects implementations [GTW10, CSDC11] and the Internet of Things [MF10, GT09] — assuming a trend implies a higher likelihood of future technology adoption). Web technologies and platforms are simple to implement as they are designed to be platform independent and have widely available implementations in most modern programming languages. Furthermore, Web security layers are a sub-

ject of extensive study and trial[ADLH+02, YWZ+04, Res01] providing a solid ground for building secure systems. In this chapter we explain some of the details, issues and limitations of our implementations aiming at evaluating the framework for practical usage, and how we integrated Web technologies with our framework concepts.

5.1 A few notes on Methodology and Notation

We use a *dot notation*, borrowed from OOP, to indicate a capability request: $\mathsf{R} \longleftarrow \mathsf{B.C(P*)}$ meaning request of capability C, on behaviour B, with parameters $\mathsf{P*}$ whose result is stored in R. The pseudo-code may be simplified on some cases (by skipping simple checks or using non-optimised procedures, for instance).

When objects are registered into a behaviour mapping (manually or through one of the discovery techniques), objects also register which other objects display the behaviour mapping behaviours they are registered in. This is how objects obtain the references to the behaviour mapping behaviours. The strategy of which behaviour mapping behaviours to get the object list from, or if one or more behaviour mapping behaviours should be requested, is defined in the internal logic of each capability and it might vary from object to object.

When a capability from a system behaviour is requested (or any other behaviour, for that matter), one or several behaviour mappings behaviours are queried in order to retrieve the list of objects which display the behaviour. Capabilities are then requested sequentially until one responds successfully. If no response is found, the "On Error" procedure (Section 4.4) is followed. Although this procedure (procedure 5.1, in this example, a capability $\mathsf{C1}$ on behaviour $\mathsf{B1}$ is requested) is followed when system behaviours communicate exchange events, for the sake of clarity we are using a simplified version (Procedure 5.2) of the following procedures in this chapter.

Procedure 5.1 Request System Behaviour's Capability — Expanded Procedure

```
URIs ← BehaviourManaging .GetBehaviourURIs("B1")
success ← false
while not success and URIs not empty do
    Result ← (URIs .Pop) .C1()
    if not Result = null then
        success ← true                          ▷ no errors
    end if
end while
if success then
    return Result
else
    return Error        ▷ will follow the On Error Procedure
end if
```

Procedure 5.2 Request System Behaviour's Capability — Simplified Procedure

```
Result ← B1 .C1()
```

5.2 Behaviour Advertising

5.2.1 Internal State

- *Behaviours* - Holds the list of behaviours that the object is able to display, according to the schema defined in Listing 5.1.

Listing 5.1: Behaviour Advertising Schema.

```
<Behaviours> ::= *<Behaviour Entry>
<Behaviour Entry> ::= BEHAVIOUR_NAME END-POINT_URI
    BEHAVIOUR_DEFINITION_URI
```

5.2.2 Events

- *NewBehaviour* - Raised to announce that a new behaviour is made public by the object.

5.2.3 Capabilities

Register Behaviour (procedure A.18) registers an object's behaviour so that it is advertised when requested or actively when discovering Behaviour Mapping behaviours.

Register Behaviour URI (procedure A.18) same as previous but registers a behaviour based on a description accessible through an external URI.

Advertise (procedure A.19) provides the list of behaviours registered in the Behaviour Adversiting behaviour.

As explained in chapter 3, in one of its modes of operation, the Behaviour Mapping behaviour is in charge of discovering and registering objects in the environment. We implemented a simple service discovery system where the Behaviour Mapping splits the network address range (depending on the network settings) in groups of a predefined size, and instantiates a discovery component with its own thread, for each of the address groups. This component works in three steps:

1. Ping each IP address in the group

2. If there is a response from a host, for each port configured for discovery send an HTTP request to http://IP:PORT/behaviouradvertising/advertise — in other words, try to request the *Advertise* capability of the *Behaviour Advertising* behaviour on that host.

3. If the host responds with a 200 code, read the response and register the object and the behaviours it advertises.

In this implementation, the *Advertise* capability responds with the list of behaviours the object displays, the URI of each behaviour definition and the behaviour's end-point URI, as shown by Listing 5.2. the end-point URI may be absent if it follows the default format described in Section 5.7.1.

Listing 5.2: Capability structured data response example.

```
{behaviours: [
  {'name': 'messagedisplayer',
  'uri':'http://10.2.61.201:9090/messagedisplayer',
  'definition': 'http://www.openobjects.com/defs/
      messagedisplayer'}
]}
```

5.3 Behaviour Mapping

The Behaviour Mapping stores a map between some or all behaviour names advertised by the objects in the environment and the corresponding object URIs, and is in charge of registry and discovery of these. This process can be either passive or active (or both) as it can behave passively, allowing objects to register themselves, and actively discovering objects and querying for their behaviours. It is also used to retrieve object URIs by behaviour name, and the list of all registered behaviours.

Objects can be ranked according to their availability and past performance. An object rank is a number, higher ranking meaning being preferred to be selected. In our implementation, Behaviour Mapping checks the availability of objects when behaviours are requested (procedure A.6), penalises (moves them down the list) those that are not available and returns those that are. When new behaviours are registered they are initially added to the top of the list (high ranking).

When an object (and its behaviours) are registered, the Behaviour Mapping may optionally request a subscription for all the events related to the behaviours displayed, automatically on behalf of the object.

5.3.1 Internal State

- *BehaviourDatabase* - A map of behaviour names and a list of the URIs of the objects that display them. The order in which the objects appear on the list is related to their ranking. This object follows the schema specified in Listing 5.3.

Listing 5.3: Behaviour Mapping Database Schema.

```
<Behaviour Database> ::= *<Behaviour Entry>
<Behaviour Entry> ::= BEHAVIOUR_NAME *<Object>
<Object> ::= OBJECT_URI <Active> <Ranking> <Manual Ranking>
<Active> ::= True | False
<Ranking> ::= NUMBER
<Manual Ranking> ::= True | False
```

5.3.2 Events

- *BehaviourActivated* - Raised during the activation of a new behaviour-object pair. Has two parameters: *BehaviourID* and *ObjectURI*;

- *BehaviourDeactivated* - Raised during the deactivation of a behaviour-object pair. Has two parameters: *BehaviourID* and *ObjectURI*;

5.3.3 Capabilities

Register Behaviour capability (procedure A.1) registers a new object and a behaviour that the object displays to the Behaviour Mapping behaviour. It takes a BehaviourID, an ObjectURI and a SubscribeToEvents flag. Adds a new tuple that relates the BehaviourID and a list of objects, if there is not one yet, then adds the object to the list. If the SubscribeToEvents option flag is chosen, it adds a subscription targeting the object for all events related to the behaviour.

Deregister Behaviour (procedure A.2) performs the opposite operations of Register Behaviour: it deregisters the object from the list of objects that display the given behaviour and raises a BehaviourDeactivated event.

Activate Behaviour (procedure A.3) sets an *active* flag for that object-behaviour pair to true.

Deactivate Behaviour (procedure A.4) sets an *active* flag for that object-behaviour pair to false.

Rank (procedure A.5) defines a manual ranking for an object in order to force the usage of an object when requesting a behaviour, or on the other hand, to set a low ranking for an object that is considered last resort — to be used when there are no other available objects displaying a given behaviour.

Procedure 5.3 Behaviour Mapping — Get Behaviour URIs

Input: BehaviourID
Output: ObjectURIList

for all Object in BehaviourDatabase .Get(BehaviourID) sorted by descending ranking **do**
  ObjectURIList ← new List
  if Object .Uri is not accessible **then**
    if Object .Uri is not Object .ManualRanking **then**
      Penalise(Object)
    end if
  else
    ObjectURIList .Add(Object .Uri)
  end if
  return ObjectURIList
end for

The **Get Behaviour URIs** capability (procedure A.6/A.10) takes a Behaviour ID and responds with a List of URI's of objects that display a specific behaviour. Each of the URI's are first tested to check if they are currently available. If they are not and the object is not manually ranked, the respective object penalised in terms of ranking (Penalise function), so that the list tends to have the URI's in order of the frequency of their availability. If the process of checking availability takes too long or if objects are known to be reliable and/or requested often, this check may be scheduled independently of the execution of this capability.

5.4 Event Broker

An event broker is capable of receiving events and forwarding them to objects who are targeted by the event or subscribed to it. It is also responsible to query the rule repositories to check if there is a rule that applies to the event and if so, request its execution from the Rule Execution. Event brokers follow a Publish/Subscribe model[EFGK03] where objects can subscribe to events based on Event Type selectors (as specified in Section 4.4).

5.4.1 Internal State

- *Subscriptions* - a map between an event selector and an object URI, following the schema in Listing 5.4

- *EventHistory* - Holds a history (a plain list) of past events.

Forward Event (procedure 5.4/A.14) forwards events to its subscribers and requests the execution of rules that are triggered by the event. The outcome of the execution of the rule (or the execution of the last rule, in case several rules match the same event) is returned; this allows for rules to use this behaviour to include in their logic outcomes from other rules.

Subscribe (procedure A.15) registers an object's subscription to events using an event selector.

Unbscribe (procedure A.16) deregisters an object's subscription to events using an event selector.

Listing 5.4: Event Subscriptions Schema.

```
<Subscriptions> ::= *<Event Entry>
<Event Entry> ::= EVENT_SELECTOR *<Object>
<Object> ::= OBJECT_URI
```

5.4.2 Capabilities

Procedure 5.4 Event Broker — Forward Event

```
Input: InEvent
Output: Outcome
                ▷ Store the Event in EventHistory (maintain max # of stored events)
if EventHistory.Length > MaximumHistoryLength then
    EventHistory .Remove(0)
end if
EventHistory .Add(InEvent)
                                                ▷ Send Event to the Recipients
for all Recipient in InEvent .To do         ▷ External recipients may require
parameters to be inserted into the recipient URI
    Replace Parameter values in Recipient
    Send InEvent to Recipient
end for
                        ▷ Look for subscribers of this event and forward it to them
if inEvent .Visibility is Public then
    Subscribers ← Subscriptions .GetMatches(InEvent)
    for all Subscriber in Subscribers do
        Send InEvent to Subscriber
    end for
end if
rules ← RuleStorage .GetRulesByEventType(InEvent)
                                                ▷ Request Rxecution of Rules
for all Rule in Rules do
    Outcome ← RuleExecution .Execute(Rule)
end for
return Outcome
```

5.5 Rule Repository

Rule Repository is the System Behaviour in charge of storing the rules of a system. It's storage is based on a mapping of the event types and the rules that respond to them. Rules can be accessed by either using the *Get Rule* capability

or by appending the Rule ID to the URI of the Rule Repository (for example: http://192.168.0.23/rulerepository/ruleid)

5.5.1 Internal State

- `Rules` - A map of event selectors (as specified in Section 4.4) and rules that are triggered by them. Rules are objects containing an identifier (ID), the rule code (code) and a boolean state indicating whether the rule is active or not (active). This object follows the schema specified in Listing 5.5.

Listing 5.5: Rule Repository Schema.

```
<Rules> ::= *<Event Entry>
<Event Entry> ::= EVENT_SELECTOR *<Rule>
<Rule> ::= RULE_ID RULE_CODE <Active>
<Active> ::= True | False
```

5.5.2 Capabilities

Register Rule (procedure A.8) registers a new rule in the Rule map.

Deregister Rule (procedure A.11) deregisters a rule from the Rule map.

Activate Rule (procedure A.11) activates an existing rule on the Rule map.

Deactivate Rule (procedure A.11) temporarily deactivates an existing rule from the Rule map.

Procedure 5.5 Rule Repository — Get Rules By Event

```
Input: Event
Output: EventRules

EventRules ← Rules .Get(Event.type)
for all rule in EventRules do
    if not rule.active or ParseCondition(rule.condition, Event) is false then
        EventRules .Remove(rule)
    end if
end for
return EventRules
```

Get Rules By Event (procedure 5.5/A.12) returns a list of active rules

which are triggered by the provided event type and whose conditions are true. The ParseCondition function (implementation specific, not specified here) parses and validates the condition expression in the rule, and returns a boolean value corresponding to the outcome of the expression. The rule is only returned if the expression is true.

5.6 Rule Execution

Rule Execution is the behaviour in charge of interpreting rules and producing the necessary events during execution.

5.6.1 Events

- `ExecutionError` - Raised during the execution in case of an error.

5.6.2 Capabilities

Procedure 5.6 Rule Execution — Execute

```
Input: Code
Output: Response

for all Instruction in Code do
    if Instruction .Execution is Bottom-up then
        for all Parameter in Instruction .Parameters do
            if Parameter .Value is Instruction then   ▷ execute instructions in parameters recursively
                Parameter .SetValue( Execute(Parameter .Value))
            end if
        end for
    end if                                            ▷ simplified capability request.
    return Response ← Instruction .Request .Behaviour .Capability   ▷ simplified request capability in the parameter
end for
return Response
```

Execute (procedure 5.6/A.7) Receives and delegates the execution of a piece of rule code. This piece of code can be an entire rule, a single instruction (a

capability request) or a sequence of instructions. An instruction can be in the form of a tree, with parameters linked to a series of secondary instructions. The event that triggers the rule may have a number of properties. These properties can be accessed in the rule condition or as parameter values of the capability requests, in the "actions" field of the rule. In order to access an event property, we use the syntax below. We use the dollar sign ($) to start a property identifier and backslash (\) as the escape character — in order to write a dollar sign inside a string we use: \$.

$BEHAVIOUR_NAME.EVENT_NAME.PROPERTY_NAME

Before executing a rule, Rule Execution replaces the references to properties (in the above format) with their values defined in the event. Examples of using event properties in rules can be seen in Chapter 6.

An instruction can be executed in either a *bottom-up* or *top-down* manners. If the instruction is set to be executed as bottom-up, the secondary instructions passed as values on the parameters should be executed first (in a process that might be recursive if these are also meant to be executed bottom-up) and their results fed to the parameters on the parent instruction and passed on to the respective capability. If otherwise an instruction is set to be executed top-down, the whole tree of instructions is passed on to the capability and it is up to the capability itself to 'decide' when, if and how many times the instructions linked to its parameters should be executed.

As an example, the rule in Listing 5.6 indicates that, assuming the conditions hold, the Rule Execution should request the execution of the *SetSoundVolume* capability of the *Television* behaviour. According to the rule, the volume parameter is attached to a secondary *Addition* capability request whose output feeds into the value of volume.

On a bottom-up approach (Listing 5.1), the Execution requests the secondary capability first — E1 — (thus starting at the bottom of the tree) and then replaces its request with the output value when requesting the *SetSoundVolume* capability — E2 — in a process we can *parameter pre-processing.*

On the other hand, on a top-down approach (Listing 5.2), the Execution sends the whole request — E1 — to the capability, without parameters being pre-processed. It is then up to the capability to make the secondary requests (potentially in a recursive way). The capability may, for example, only make the

request if some condition applies, as in Listing 5.3. In this example, a conditional capability (requested by E1) follows one path of execution or another, depending on the outcome of the *condition* parameter.

The Open Objects framework supports both methods of execution as both have different advantages and disadvantages. Top-down provides the capability with finer control over the execution which makes it possible to have, for example, capabilities that perform cycles or conditional branching. On the other hand, bottom-up execution is less flexible but more friendly towards lighter weight objects as by pre-processing the parameters, the Execution sends a request that is smaller and simpler to execute.

Listing 5.6: Rule example.

```
{
  "rule": "R3",
  "event": "BabyMonitor.soundDetected",
  "conditions": "volume > 5",
  "actions": [
      {
          "event": "Television.SetSoundVolume",
            "volume":
      {
          "event":"Math.Addition",
          "A": 5,
          "B": 20
      }
    }
  ]
}
```

5.7 A RESTful approach

As the framework's implementation follows a Service-Oriented[Erl08] approach over Web technologies, a number of solutions is available. Most of these solutions (notably [CDK+02]) wrap messages in custom message envelopes, which are then wrapped in HTTP[FGM+99] and TCP/IP[FS11] envelopes, which takes unnecessary bandwidth for our needs. On the other hand, RESTful[Cos07] ser-

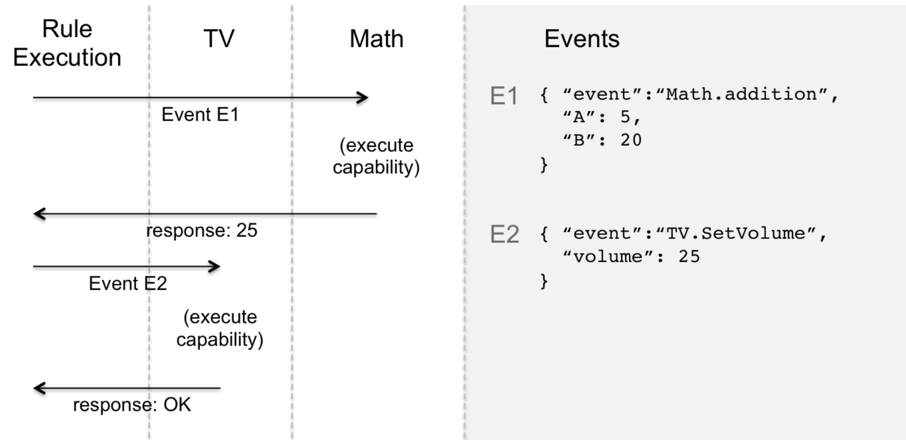

Figure 5.1: Bottom-up execution. Events attached to parameters are pre-processed before being sent to capabilities. (Names shortened for convenience)

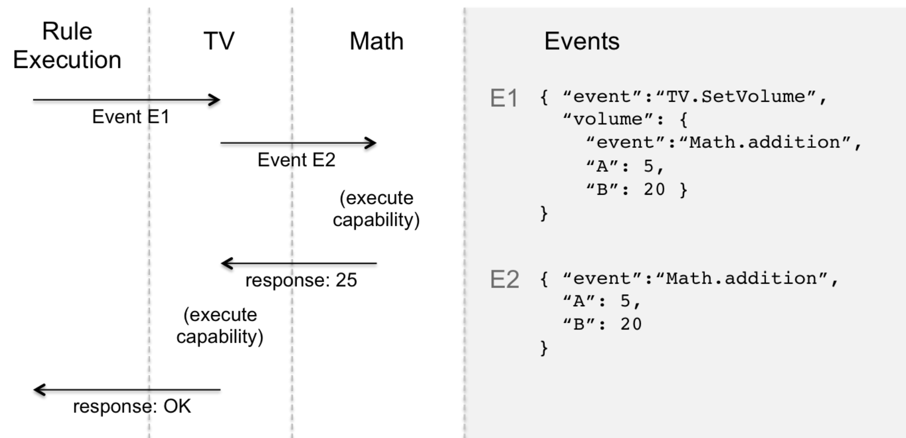

Figure 5.2: Top-down execution. Capabilities are sent full events before the parameters have been pre-processed and it is up to them to decide if/when/how many times to request the capabilities attached to the event parameters. (Names shortened for convenience)

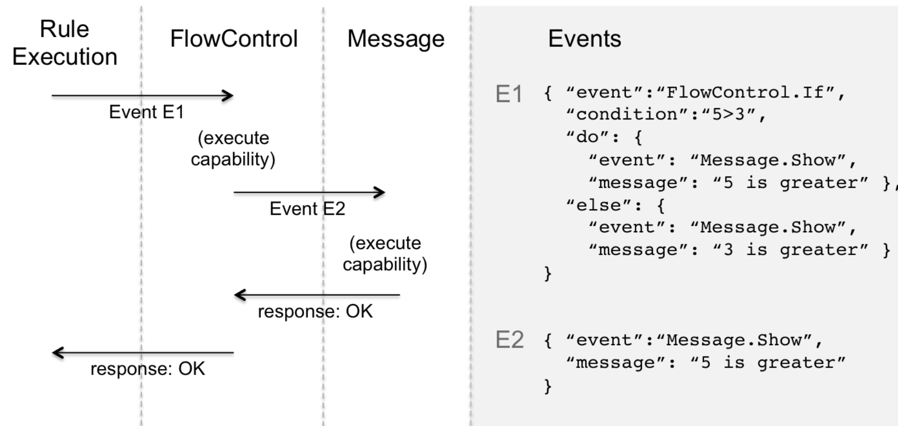

Figure 5.3: Another example of top-down execution. In this case not all branches are executed and it is up to the If capability to choose which capability attached to its parameters should be requested. (Names shortened for convenience)

vices (which we chose to use for this implementation) reuse the HTTP envelope and methods for their own purposes, making a more efficient use of bandwidth.

Alternatively the framework could be implemented using CoAp[CoA12] — a secure communications protocol that is lightweight and efficient when compared to HTTP, specifically designed for Machine-to-Machine and Internet of Things applications where devices are often constrained in terms of memory and processing power. We chose to use HTTP for our prototype implementation due to the wider availability of software programming libraries and ease of use when working with external we services. We are assuming that from the point of view of a proof-of-concept, if the framework works over HTTP, it would also work over CoAp equally well, with the addition of the security layer implemented into the protocol.

A RESTful service is composed of a set of resources, each with a unique URI. Each resource can be requested using an HTTP method (typically GET, PUT, POST and DELETE), a set of parameters (usually paremeters follow the URI and are separated by "/") and a message body. A GET request indicates a request for information; PUT typically updates a resource with information on the message body; POST is usually used to add new data to a resource; DELETE, as the name implies, removes an entry from the resource.

5.7.1 Capabilities as Resources

In this implementation, the system is connected to a local Ethernet[MB76] network and each object is uniquely identified by its URI. We chose to use port 9090 as the default to communicate between objects (although several ports can be specified for Behaviour Discovery). We also consider each behaviour displayed on each object to be a single RESTful web-service, with its own end-point, typically of the format:

http://OBJECT_URI/BEHAVIOUR_NAME/

and each of the behaviour's capabilities to be a single resource on the service, accessible through:

http://OBJECT_URI/BEHAVIOUR_NAME/CAPABILITY_NAME/

There is a variety of possible ways of requesting and passing parameters to a REST resource, e.g. query parameters, path parameters, HTTP headers, etc. In this implementation we chose to define that all requests are made using a POST method with the event (as defined in Listing 4.9) as the message body. Although this approach isn't standard and some might argue that it isn't strictly speaking RESTful, we chose it as we wanted to standardise the capability requests in order to make the Rule Execution content and semantics agnostic.

Requesting Capabilities

A typical capability request looks like the HTTP request on Listing 5.7.

Listing 5.7: Capability HTTP Request event example.

```
POST /television/setsoundvolume/ HTTP/1.1
Host: 192.168.0.45:9090
Content-Type: application/json; charset=utf-8
Content-Length: 150

{
  "event": "Television.SetSoundVolume",
  "id": "e239",
  "from": "192.168.0.42",
  "to": "192.168.0.45",
  "cid": "4843",
  "volume": 25
}
```

Capabilities may respond to the request with one of the following HTTP response codes:

200 OK - when the capability was successfully requested and executed;

400 - Bad Request when the request was not made properly, for instance a required parameter was not provided;

404 Not Found - when the request is made to an object that does not possess the requested capability;

500 Internal Server Error - when there is a capability execution error.

A typical capability response looks like the Listing 5.8 for single value responses or like Listing 5.9 for structured data responses.

Listing 5.8: Capability single value (number 15) response example.

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 2

15
```

Listing 5.9: Capability structured data response example.

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 33

{"values": [0, 1, 1, 2, 3, 5, 8]}
```

This method is, however, limited by the request timeout — the amount of time that the client (the object that requests the capability) waits for the server (the object where the capability is located) to respond. There is an alternative, more flexible approach to responses: instead of returning the output of a capability execution directly through the HTTP response following the initial request, a capability may respond initially with a "200 OK" and respond later with another event sent to the initial request's sender, with the initial event's id in the "in response to" field defined in Listing 4.9.

5.7.2 Web Client and Server as Actuator and Sensor

In order to be able to receive and send events, objects need to implement a Web server, a Web client and a sispatcher. The Web client, however, is not required in the case of very simple objects that do not initiate requests. A Web server receives the request from a client and hands it to the dispatcher. The dispatcher then hands the event to the correct internal component or function depending on the capability specified by the request path and based on an internal predefined mapping. If the object does not implement the requested capability, the Dispatcher tells the Web server to respond with a 404 error.

5.8 Experimental Implementation Platforms

During our evaluation and in order to produce the prototypes presented in Chapter 6 three different implementations of Open Objects have been carried out, with different levels of complexity and completeness. All three implementations were then integrated into the same system as an example of interoperability.

We implemented a non-optimised complete Open Object in **Java**[1] that runs on a desktop computer. All system behaviours were instantiated in this imple-

[1] http://www.java.com/

mentation, which means it can be run independently. For data storage we used non-persisting in-memory java object databases to support behaviour states. Two side projects spawned from this implementation, both focused on simplicity of use and lightweightness. One is called SimpleREST and contains a small, easy to set up web server and a request dispatcher allowing for Java objects to subscribe to URL patterns (including URL parameters). The second project is called SimpleWebClient and, as the name implies, it makes it simple to perform different kinds of HTTP requests.

In order to evaluate the feasibility of implementing an Open Object firmware on a small lightweight device, we developed a C version for **Arduino**[2] with a stripped down web server and web client, a Behaviour Advertiser and Domain Specific Behaviours that interact with hardware sensors and actuators. We used a variant of an Arduino called Nanode[3] that runs on a 16MHz ATmega328P processor chip with 32Kb SRAM. The compiled code and memory needed for runtime occupy just under 32Kb. The Arduino has a few limitations due to its simple architecture, for instance, it's a single threaded environment, therefore only one request can be handled at a time. Also, due to its small memory capacity, only relatively small rules could be sent to the Arduino. This limitation showed us that we needed two execution modes: a top-down and a bottom-up (as explained in chapter 4). In the bottom-up execution mode, the capability requests connected to each request parameter are executed first so that the parameter values are replaced with the output of each secondary capability request, resulting in a much smaller request size, suitable for lightweight devices.

We also implemented a **Python** version of an equivalent object, with the same behaviours, running on a desktop computer to test the resilience[RSP11] of functionality in the system — maintaining functionality by handing over part of the execution to another device when one becomes inaccessible. On this test, the Python Open Object would take over the Arduino functionalities when the latter was turned off.

[2] http://www.arduino.cc
[3] http://www.nanode.eu

5.9 Summary

In this chapter we proposed a specification for the implementation of each of the System Behaviours and their events and capabilities by defining and explaining their procedures. We also explained some implementation details, issues and limitations that influenced the methodology adopted in the experiments described in Chapter 6. We detailed how to connect objects in this implementation and how behaviours and capabilities are matched to web-services and resources and how these are discovered and registered by the Behaviour Mapping behaviour in practice. We also gave an overview over the implementation over three different platforms, which are shown working together in the next chapter.

6 Case-Studies

In this chapter we demonstrate a number of experiments that were carried out using the implementation described in Chapter 5. For each experiment, we discuss its aims, what the trial is testing or proving, the methods, techniques and assets used, the results and our conclusions/outcomes. The chapter is organised so that in each experiment a new concept or methodology is incrementally introduced. We also identify certain limitations of the implementation and propose strategies or methods to overcome them.

6.1 Smart Lucky Waving Cat

For this preliminary experiment, we developed a system to augment a chinese lucky waving cat, pictured in Figure 6.1, during an Internet of Things hacking event (which earned the best Nanode[1] project prize). The idea behind this prototype was to create a household object that provides useful information by

[1] http://www.nanode.eu

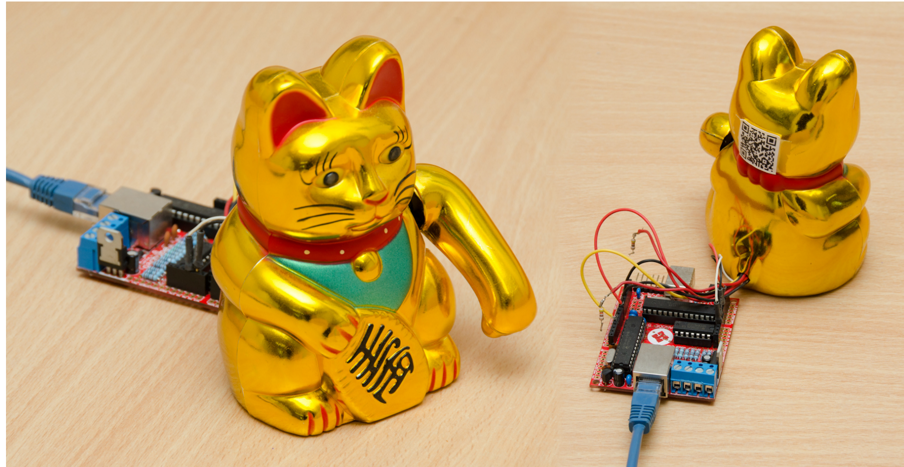

Figure 6.1: Smart Lucky Waving Cat

looking at it. We aimed at testing the practicability of exposing capabilities as web services on a decorative object embedded with lightweight electronics.

6.1.1 Methodologies

We fitted the object with an Ethernet connected Nanode Arduino[2] clone, LED lights in its eyes, a stepper motor on its arm and an internal speaker, each of these being a capability of the object. This was carried out with a preliminary design of the framework, for example, the concept of behaviours, for example, was not present. We also developed an external web service as a gateway for information on the internet such as Twitter feeds and weather conditions. A rule (using a preliminary rule syntax, different from the one explained in Chapter 4) requests the external services and exposes a capability that provides digested values, which in turn is queried by the Nanode and changes its capabilities accordingly. The user is able to configure the Web service by scanning a 2D barcode located on the back of the object, allowing them to choose, for example, what Twitter search terms affect each of the object's capabilities.

As an example of a rule, we used the arm movement to illustrate the wind speed in the current location, the speaker to play different songs when certain expressions were mentioned in Twitter (our account name or "royal wedding" for example) and the LED lights on the eyes lighting up when a certain hashtag

[2]http://www.arduino.cc

was used on Twitter — in this example, the eyes would become more lit with red when someone wrote the word *evil* on Twitter and less when someone wrote *good*. The cat was also able to perform certain programmed choreographies involving all the actuators, triggered by any of the previously mentioned event.

6.1.2 Results and Conclusions

This preliminary case study showed that it is feasible and useful to place a Web client and Web server on a lightweight physical object and exposing its capabilities to the outside, augmenting its purpose and allowing the user to change its behaviour. This case study also made us understand that, although basic event-condition-action rules can be applied to a variety of scenarios, more algorithmically complex rules could open doors to more complex and mature application behaviours. Over the next next case studies we introduce a number of additions to the framework that allowed us to target more complex scenarios.

6.2 Algorithmic Test

With this short set of tests, we aim at testing whether basic algorithmic constructs — arithmetical and boolean operations, cycles and conditional branching — can be supported by (a) the expressiveness of the rule language, (b) the execution strategies and (c) the System Behaviour interactions.

6.2.1 Methodologies

For this experiment we used a single complete Open Object (all system behaviours displayed by a single object) with a Java implementation on a desktop computer. We wrote two rules, on Listings 6.1 and 6.2, to be triggered independently during the tests. Note that we did not implement parsing of arithmetical and boolean expressions directly. Instead, we developed a *Math* behaviour with capabilities for each operation, e.g. in R7.1 we use *Math.Addition* and in R7.2 *Math.GreaterThan*.

Listing 6.1: Algorithmical Test Rule R7.1

```
{
  "rule": "R7.1"
  "event": "Test.PleaseBeep",
  "actions": [
  {
    "event": "FlowControl.Repeat",
    "n":
    {
      "event": "Math.Addition",
      "A": 1,
      "B":
      {
        "event": "Math.Addition",
        "A": 2,
        "B": 1
      }
    },
    "do":
    [
      {
        "event": "FlowControl.Sleep",
        "t": 1000
      },
      {
        "event": "Sound.Beep"
      }
    ]
  } ]
}
```

In Listing 6.1, an event *Test.PleaseBeep* is produced by the test application, which triggers rule R7.1. The rule requests the execution of *FlowControl.Repeat*, which requires two parameters: “n”, which indicates how many times to repeat the cycle; and “do” which indicates what actions are to be performed on each cycle. *Math.Addition* sums the values specified in “A” and “B”, and returns

the result. In R7.1, a result of a *Math.Addition* capability request is fed into another *Math.Addition* and the value returned by the latter is 4, hence causing the actions in specified in "do" to be repeated four times. When *FlowControl.Repeat* repeats each cycle, it requests the execution of two capabilities: *FlowControl.Sleep* (with a parameter "t" of 1000 milliseconds or 1 second) which causes the execution to stop for one second and *Sound.Beep* which produces an alert sound.

Listing 6.2: Algorithmic Test Rule R7.2

```
{
  "rule": "R7.2"
  "event": "Weather.WeatherChanged",
  "actions": [
  {
    "event": "FlowControl.If",
    "condition":
    {
      "event": "Math.GreaterThan",
      "A": $Weather.WeatherChanged.temperature,
      "B": 20
    },
    "do":
    {
      "event": "Message.Display",
      "message": "It's warm"
    },
    "else":
    {
      "event": "Message.Display",
      "message": "It's cold"
    }
  } ]
}
```

in Listing 6.2, an event *Weather.WeatherChanged* is produced by the *Weather* behaviour running in the test application, which triggers rule R7.2. This rule uses the *FlowControl.If* capability to compare the "temperature" parameter of

the *Weather.WeatherChanged* event with the fixed value of "20". As we have not implemented parsing of boolean expressions, we used the *Math.GreaterThan* capability to compare the two values and return "true" or "false" according to the result. The *FlowControl.If* capability requests the execution of the "do" or the "else" branch, according to the outcome of the "condition" parameter value.

6.2.2 Results and Conclusions

Both rules R7.1 and R7.2 were executed successfully after the events were produced. Summarised (and made more accessible to a human reader) system trace of the rules execution follows. Certain repetitive steps are not shown here, such as the full sequence of requesting a capability:

1. Behaviour B1 on object OBJ1 wants to request capability C1 of behaviour B2;
2. B1 sends event B2.C1 to Event Broker through the Web Client on OBJ2
3. The Web Server on the Event Broker's object receives event B2.C1;
4. Dispatcher forwards it to Event Broker;
5. Event Broker requests the URI of the object that displays B2, from Behaviour Mapping;
6. Behaviour Mapping responds with the URI of OBJ2
7. Event Broker forwards B2.C1 to OBJ2;
8. Web Server on OBJ2 receives event B2.C1;
9. Dispatcher sends the event to the behaviour B2.
10. B2 executes capability C1 and responds with its outcome, to Event Broker;
11. Event Broker forwards the response event back to B1 on OBJ1.

System trace of Rule R7.1:

1. Web Server started;
2. System, Math, FlowControl and Sound behaviours are registered in the Behaviour Mapping;
3. Rule R7.1 is registered in Rule Repository;
4. Web Server receives event Test.PleaseBeep;
5. Dispatcher sends the event to Event Broker;
6. Event Broker requests rules that apply to event from Rule Repository;

7. Rule Repository finds rule R7.1 and responds by sending it back to Event Broker;
8. Event Broker sends rule to Rule Execution;
9. Rule Execution reads the first action on the rule (FlowControl.Repeat);
10. Execution mode is not specified, so the default "Top-down" mode is used;
11. Execution asks Behaviour Mapping which object displays the FlowControl behaviour and receives the URI of the object;
12. The action (the Repeat capability request) is sent to FlowControl;
13. The internal algorithm (machine code) of the Repeat capability requests the value of "n";
14. The internal algorithm of the Repeat capability requests execution of FlowControl.Sleep, with the parameter "t" of 1000;
15. Sleep pauses the execution for one second by taking said time to respond to the request;
16. Sleep responds, after one second, back to Repeat with OK (capability execution succeeded);
17. Repeat requests the execution of the Beep capability on the Sound Behaviour;
18. The Beep capability produces a sound and responds back to Repeat;
19. The Repeat capability repeats the previous four steps four times;
20. The Repeat capability responds back to Execution with OK;
21. Execution of R7.1 terminates.

During the execution, we could perceive a notification sound being played four times, with a one second interval. The experiment shows (a) interaction of the System Behaviours on a Complete Object, (b) forwarding and dispatching of events, (c) Top-Down execution of cycles in capabilities and (d) execution of chains of capability requests (one output feeds into another's input).

System trace of Rule R7.2, further simplified for clarity and economy of space:

1. Web Server started;
2. System, Math, FlowControl and Sound behaviours are registered in the Behaviour Mapping;
3. Rule R7.2 is registered in Rule Repository;

4. Weather behaviour produces a WeatherChanged event when the weather condition changes;
5. Event Broker receives WeatherChanged event, with parameter "temperature" set to 22;
6. Event Broker requests rule triggered by WeatherChanged from Rule Repository;
7. Event Broker received R7.2 and sends it to Rule Execution;
8. Rule Execution replaces references to event parameters in the rule with their values (in this case it replaces Weather.WeatherChanged.temperature with the value specified in the event: 22);
9. Rule Execution send the FlowControl.If capability request event to FlowControl behaviour;
10. FlowControl requests the Math.GreaterThan capability from Math behaviour;
11. the GreaterThan capability compares parameters "A" and "B" (in this case 22 and 20) and responds with "true";
12. The If capability received the outcome "true" of the GreaterThan capability execution;
13. as "condition" has the value of "true", the "do" branch of the rule will be requested for execution, by the If capability;
14. The Display capability of the Message behaviour is requested, with the parameter "message" set to "It's warm";
15. Execution of R7.2 terminates.

The Weather behaviour is programmed to check the local weather at regular intervals and, when there is a change, it produces a *WeatherChanged* event, which triggers rule R7.2. During the execution, the Message behaviour displayed one of the two different messages on the computer screen, according to the current temperature. This experiment shows (a) conditional branching of the execution — for example, a capability choosing to execute one branch over another depending on the value of a parameter — and (b) the use of event parameters (the use of the temperature parameter).

Both examples show the main advantage of top-down execution: giving control to the target capability over if, when and how often to request the capa-

Figure 6.2: Open Power Socket

bilities attached to its parameters which, for example, this makes it possible to have cycles and conditional branching.

6.3 Open Power Socket

We modified a common power socket in order to control the electric appliances connected to it (such as lamps, illustrated in Figure 6.2). The aim here was to test the framework implementation in a real world scenario, with a physical object that exposes a capability to the outside. As our current implementation relies on maintaining open network connections (with a predefined maximum time duration) throughout the execution of a rule, we wanted to develop a way of overcoming this limitation in order to have rules running over a prolonged period of time. Our objective for this case-study was to develop a simple rule that switched on and off a lamp plugged into the extension, at random periods of time, to simulate a human presence in an empty house. As an example of the applicability and flexibility of this setup, on another instance, we wrote a rule that turned on and off a fan depending on the local weather.

6.3.1 Methodologies

We augmented a power socket extensions with a relay and a network (Wi-Fi) capable Arduino (powered by the extension), and developed a generic behaviour called *Device* which shows generic capabilities present in most devices. On this behaviour we implemented capabilities *TurnOn*, *TurnOff* and *IsOn*, which switches the relay on and off, and checks if the device is on. In order to support our objective of switching the lights on and off for random periods of time, we wrote the rule in Listing 6.3.

Listing 6.3: Light Scheduler Rule

```
{
  "rule": "LightScheduler"
  "event": "Test.ScheduleList",
  "actions": [
  {
    "event": "FlowControl.If",
    "condition":
    {
      "event": "Math.GreaterThan",
      "A": { "event": "Time.Now" },
      "B": { "event": "State.Get", "name": "timeout" }
    },
    "do":
    [
      {
          "event": "FlowControl.If",
          "condition": {"event": "Device.IsOn"},
          "do": {"event": "Device.TurnOff"},
          "else": {"event": "Device.TurnOn"},
      },
      {
        "event": "State.Put",
        "name": "timeout",
        "value":
        {
```

```
            "event": "Math.Addition",
            "A": { "event": "Time.Now" },
            "B":
            {
                "event": "Math.Random",
                "min": 60,
                "max": 6000
            }
          }
        }
      ]
    },
    {
      "event": "Parallel.Split",
      "do":
      [
        { "event": "FlowControl.Sleep", "t": 5000 },
        {
          "event": "EventBroker.ForwardEvent",
          "inEvent": {"event": "Test.ScheduleList"}
        }
      ]
    } ]
  }
```

This rule runs in a loop, defining a time in the future. When the time is reached, the state of the connected device is inverted and a new future time is set.

Given a *Test.ScheduleList* event, the *LightScheduler* rule checks if the current time is larger than the *timeout* time stored in the *State* behaviour (which will be true for the first run as there is no value stored). If the current time is larger, the state of the device is inverted and a new timeout is generated based on a random number of seconds between 60 and 6000. The last step of the rule is a *Parallel Split* which creates an independent branch of execution (and effectively terminating the main branch, as there are no more actions to be executed in it). On this branch, a *Sleep* capability is requested (in order to stop the execution

for 5 seconds) and then a new *Test.ScheduleList* event is produced and sent to the Event Broker, resulting in a new execution of the rule. If we did not use the *Split* capability, the initial HTTP request for the execution of the rule would timeout as we would be continuously requesting new capabilities recursively (similar to a stack overflow computing error).

When developing the prototype for this experiment we implemented three supporting domain independent behaviours:

- *Time* - a *Now* capability provides the current time (in UNIX timestamp format);
- *Parallel* - with a *Split* capability that split the execution in order to have an independent branch;
- *State* - Able to store and retrieve values in memory, accessible by a key name, using the *Get* and *Put* capabilities.

These and the system behaviours were implemented on the desktop machine. The Open Power Socket displays *Device* and *Behaviour Advertising* behaviours (discovered automatically by Behaviour Mapping).

6.3.2 Results and Conclusions

Through the lifetime of the system, the rule runs continuously as expected, and the lights are turned on and off at random intervals. This case-study showed us that by allowing rules to be executed synchronously over an HTTP connection brings some advantages — for instance, we can produce an event that triggers a rule and receive the outcome of the rule as a direct response of the event request, effectively creating a high-level capability — but also the disadvantage of being limited by the HTTP connection timeout. In order to request a rule to be executed asynchronously, we implemented a Parallel Split[3], which we then use to terminate the execution of the main branch (closing the HTTP connection) and initiate a new, independent one.

We have written in the rule the basic mechanics of a scheduling system, which we could encapsulate into a reusable *Scheduling* domain independent behaviour, with a *Schedule* high-level capability that registers a time to which a given custom event would be produced. For instance, we could request the

[3] http://www.workflowpatterns.com/patterns/control/basic/wcp2.php

Schedule capability to produce custom event X at time Y, and write a rule that is triggered by X. The specification and development of different domain dependent and independent behaviours are, however, outside the scope of the framework and are presented here as building blocks developed for case studies. Also, the mechanism we use to allow a rule to continuously run despite the connection timeout could also be encapsulated in order to be easily reusable.

6.4 Resilience Test

In this small study we wanted to assess the ability of the framework to maintain service resilience, with preferred objects displaying certain behaviours. Service resilience was an important factor of a parallel project we were involved in[RSP11] and some of the concepts developed in the context of this thesis were inspired by it, including service resilience, the Behaviour Mapping and the Event Broker. We developed a prototype for displaying a message to the user using one rule and the message would be displayed on one of three Open Objects, depending on their availability and their ranking.

6.4.1 Methodologies

For this experiment we implemented the same behaviour *Message* on three different objects and platforms: an Arduino with an LCD screen, a Java application on a desktop computer and a Python script on a laptop computer. The *Message* behaviour has one capability called *Display* which shows the *message* event parameter, when requested.

On a side note, in our prototype implementation of the framework, we are using IP addresses as a means of uniquely identifying objects. This is merely a convenience and would not suit an end-product as IP's may change between executions.

We used the rule in Listing 6.4 used on this experiment.

Listing 6.4: Service Resilience Rule

```
{
  "rule": "ResilientMessage"
  "event": "Test.Resilience",
  "actions": [
  {
      "event": "BehaviourMapping.Rank",
      "behaviour": "Message",
      "objectURI": "192.168.0.2",
      "rank":3
  },
  {
      "event": "BehaviourMapping.Rank",
      "behaviour": "Message",
      "objectURI": "192.168.0.13",
      "rank":2
  },
  {
      "event": "BehaviourMapping.Rank",
      "behaviour": "Message",
      "objectURI": "192.168.0.45",
      "rank":1
  },
  {
    "event": "Message.Display",
    "message": "Hello!"
  }
}
```

In this rule we manually set the ranking of the three objects in regards to the *Message* behaviour, so that the Arduino (identified by it's IP address: 192.168.0.2) takes precedence over the laptop (192.168.0.13) and the laptop over the desktop (192.168.0.45), in terms of which object should be requested regarding that behaviour.

6.4.2 Results and Conclusions

At the start of the experiment, the objects are automatically discovered and registered by Behaviour Mapping. In order to force the execution of the rule, we sporadically produce the *Test.Resilience* (using a test HTTP request client), plug and unplug the Arduino and the laptop from the network, and observe the results. We observed that, as expected, when the Arduino is plugged in, it is discovered and when the rule is executed, the Behaviour Mapping checks that the object is available and the *Message.Display* is sent to it. When the Arduino is not plugged in, the event is sent to the next object with higher rank — the laptop, and similarly, if the laptop is not available, the event is sent to the desktop. When we plug the Arduino back in, for example, the events are once again sent to it, and we can see the message being displayed on its LCD screen.

This experiment shows two things, a simple mechanism for guaranteeing resilience of functionality, and how to manually set the ranking of an object in regards to a given behaviour.

6.5 Office Availability Device

For this experiment we developed and implemented a real-world scenario where the user develops a system to inform work colleagues or students of their availability in the office. The idea is to have a small electronic device at the door (the same object used to display a message on the previous experiment and pictured in Figure 6.3), visible from the outside, that is capable of displaying a short message along with a small coloured light that illustrates its state. There is also a cubic device (Figure 6.4) that allows the user to set different states related to his availability according to the side to which it is turned. The cube has different messages written on each side, which correspond to different "states" of availability: "Check calendar", "Back in 5", "Out of the office", "Out for lunch", "Busy" and "Come in". The system should be as automatic and as unobtrusive as possible, therefore, in the absence of exceptions, the availability should be set accordingly to events on the user's online calendar.

The door sign and the cube are considered Hollow Objects, in the sense that they are merely building blocks to be used as part of a larger interaction of objects. By default, they do not possess a particular function. Instead, they

Figure 6.3: Open Door Sign

Figure 6.4: Open Cube Controller

provide a number of low-level capabilities that interact with their actuators.

6.5.1 Methodologies

Due to the complexity of this setup, we will use the nomenclature below to refer to behaviours (Bn), devices (Dn) and events (En).

System Behaviours

- B1: Event Broker
- B2: Rule Repository
- B3: Rule Execution
- B4: Behaviour Mapping
- B5: Behaviour Advertising

Domain Independent Behaviours

B6: Time

B7: Flow Control

B8: Parallel

Domain Dependent Behaviours

B9: Calendar Checker (checks an online calendar for appointments)

B10: Orientation Sensor (communicates the orientation of an object, in relation to the gravity referential)

B11: Message

B12: Traffic Light (controls a coloured light)

Equipment

Equipment-wise, there are three different devices in the system. The behaviours displayed by each of the devices are defined between brackets:

D1: Computer (B1, B2, B3, B4, B6, B7, B8, B9)

D2: Desk Cube (B5, B10)

D3: Door Sign (B5, B11, B12)

Events and Rules

In order to coordinate the behaviours in the system, different rules were written. Each rule is attached to an event. The events produced during the execution of the system are the following.

E1: startup

E2: do_a_cycle

E3: check_cube

E4: display_availability_from_calendar

E5: check_calendar

The following rules (and the events that trigger them) have been previously written and registered in the system.

R1: start_availability_system, on E1 (produces E2 to start the system's main rule cycle)

R2: cycle_availability, on E2 (every 5 seconds, produces two events: E3 and E2, causing R2 and R3 to execute)

R3: cube_checking, on E3 (displays a message and changes the colour of the traffic light according to the side of which the cube is turned. If the cube is unavailable or the side is the default one, the rule produces E4)

R4: display_availability_from_calendar, on E4 (changes the message and colour of the traffic light according to the state of the calendar. uses a secondary rule, R5, to check the calendar)

R5: check_calendar, on E5 (checks one particular calendar to see if there is an event at the current time)

In Listing 6.5 we present one of the rules in this application (R3 — Cube Checking), as an example of how to prioritise rules. In this rule we use the *Choose* capability of the *FlowControl* behaviour to display different messages and colours of the light signal, according to the orientation of the cube, which is given by the *Get* capability of the *Orientation Sensor* behaviour displayed by the cube object. For each *option* — or side of the cube — we send the appropriate events to the *Message* and *Traffic List* behaviours. If the side of the cube is number 6, we produce the event "display_availability_from_calendar" which triggers rule R4 (which displays a message depending on the availability stated on the online calendar).

Our objective for this application is to show a message on the door sign depending on the orientation of the cube. However, if the cube is not available (e.g. disconnected), the message should reflect the state of the online calendar. In order to achieve this, we use the *onError* parameter when requesting the *Get* capability of the *Orientation Sensor* to trigger the "display_availability_from_calendar" rule. In other words, rule R3 takes precedence over R4 — R4 is only executed if there is an error on R3. Another way of implementing this would be to use the *onError* parameter "upstream" on rule R2. In other words: R2 requests R3 and if there is an error, request R4 instead.

Listing 6.5: Rule R3 - Cube Checking

```
{ "rule": "Cube Checking"
"event": "OfficeDoorSign.Check_Cube",
"actions": [
{    "event": "FlowControl.Choose",
     "value":
     {
         "event": "OrientationsSensor.Get",
         "onError":
         {
                "event": "EventBroker.ForwardEvent",
                 "inEvent": {"event": "
                      Display_availability_from_calendar"}
          }
      }, "options": [
      {
         "value": 1,
         "actions": [
         { "event": "Message.Display", "message": "Out for
             Lunch" },
         { "event": "TrafficLight.Set", "colour": "red"} ]
     },
         {... other options for each side of the cube ...}
     {
         "value": 6,
         "actions: [
         {
             "event": "EventBroker.ForwardEvent",
             "inEvent": {"event": "
                 Display_availability_from_calendar"}
         } ]
     } ]
} ] }
```

For this application we also developed a low-level capability (written in Java) called *Get State* on a behaviour *Calendar Checker* that interacts with an on-

line calendar API and takes two parameters: "calendar" — the URL of the calendar; and "date_time" — the timestamp for which to check the availability. The capability responds with one of three states: "no event", "available" or "busy". We also defined a high-level capability (defined by the rule R5, show in Listing 6.6) called *Check Calendar* that wraps the *Get State* capability and provides the appropriate parameters.

In this case, it is more convenient to have a low-level capability because checking an online calendar is a complex task. If, otherwise, we were dealing with a simpler web service, we could have just used a behaviour description to describe it, and it could have been included in a rule seamlessly.

Listing 6.6: Rule R5 - Check Calendar

```
{ "rule": "Check Calendar"
"event": "OfficeDoorSign.Check_Calendar"
"actions": [
{
  "event": "CalendarChecker.GetState",
  "calendar": "https://www.google.com/calendar/feeds/
      e4f76qik8agu...",
  "date_time": { "event": "Time.Now" }
} ] }
```

6.5.2 Results and Conclusions

Steps 1 to 10 may happen interchangeably, although some rule execution cycles may fail until D2 and D3 are discovered because not all necessary capabilities are available in the system until then. All calls to a capability are preceeded by a call to B4 so that the caller knows which object implements the capability and how to reach it. Throughout the system trace, this process is omitted, to facilitate reading and replaced by the more human-friendly and simplified "Behaviour B1 requests the execution of B2.C1", for example.

1. The rules are written and stored in B2;
2. B1, B2, B3, B6, B7, B8 and B9 are registered locally in B4;
3. B4 looks in the network for devices that implement B5 (D2 and D3);
4. B4 discovers D2 and the B5 on D2 communicates the implementation of

B10, which is then registered on B4;

5. B4 discovers and registers D3's behaviours B11 and B12;
6. E1 is raised by D1 on startup;
7. B1 picks up E1 and checks if B2 has a rule registered to be executed after E1;
8. B2 responds with R1 and B1 sends R1 to B3 for execution;
9. R1 tells the B3 to request the B1 to propagate E2, effectively starting the system's main loop;
10. Similarly to steps 6 to 9, E2 causes R2 to execute;
11. R2 has a sequence of two instructions: request B7 for suspending the execution for 5 seconds, using its "sleep" capability; request B8 to create a new branch of execution. In this branch two events are produced: E3 and E2. producing E2 causes R2 to run in a cycle;
12. E3 causes R3 to execute. R3 uses a "choose" capability which works as a "switch" instruction in procedural languages. The objective here is to choose between a set of actions, depending on a "value" parameter of the "choose" capability. "value" is connected to the "get" capability of B10. This capability responds with an integer between 1 and 6 that correspond to which orientation the object is currently facing: X up, X down, Y up, Y down, Z up, Z down.
13. For each possible orientation a different message and light colour is displayed. This is achieved by requesting the "display" capability of B11 and the "set" capability of B12.
14. In case of failure when trying to request a capability from B10, or in case D2 is on an orientation that is considered to be default (in this case position 5), E4 is produced. This is a way of specifying that R3 has precedence over R4, or in other words: R4 should apply in case R3 does not;
15. if E4 is produced then R4 is executed. R4 says that if there is an event on the user's calendar, then his status is displayed as available in the door sign, with a green light, otherwise, the sign should state that he is away, with a red light. In order to check if there are events in the calendar, R4, raises E5 and waits for the result of the rule execution;

16. E5 causes R5 to execute, which uses the "is_busy" capability of B9 with the time given by the "now" capability of B6.

As expected, when the cube is in the default position — "Check calendar" — the door sign displays a message according to the events on the online calendar. Otherwise, when the cube is in another position, representing an exception, the sign shows the message on the chosen side of the cube. This case study shows how to use a rule that dictates precedence of other rules, by choosing which events should be produced.

Also shown here, is the usage of encapsulation of functionality in rules. In order to simplify a rule, we encapsulate specific functionality in a new, reusable rule. By adding parameters to the event that triggers said rule, we are effectively parametrising the rule execution itself. For instance, the *check_calendar* rule works as a high-level capability, by encapsulating the functionality of checking an online calendar for an even, and responding with a value of "true" or "false". This is the basis of adding new capabilities to existing objects: a rule defines a new high-level capability, a new behaviour description is written, describing that capability (and possibly others) and the behaviour description is then registered in the objects Behaviour Advertising behaviour.

6.6 Adding a new Behaviour to an Object

In order to generalise and encapsulate common actions, one can either create rules that encapsulate the desired functionalities or create a new behaviour based on high-level capabilities defined by rules. When creating a new behaviour, one must:

1. Write the rules that describe the behaviour's capabilities;
2. Register the rules in a Rule Repository;
3. Write the Behaviour Description;
4. Register the Behaviour Description on the target object's Behaviour Advertising.

As discussed in Section 6.3, we arrived at the conclusion that we could encapsulate and improve the scheduling system to make it generic and reusable.

In order to assist us in the creation of a *Scheduler* behaviour, we wrote the rule in Listing 6.8 to perform the tasks described above.

Listing 6.7: Registering a Scheduler Behaviour

```
{
    "rule": "RegisterScheduleBehaviour",
    "event": "startup",
    "actions": [
    {
        "event": "RuleRepository.RegisterRule",
        {
            "rule": "Schedule"
            "event": "Scheduler.Schedule",
            "actions": [
            {
                "event": "FlowControl.If",
                "condition":
                {
                    "event": "Math.GreaterThan",
                    "A": { "event": "Time.Now" },
                    "B": "$Scheduler.Schedule.timeout"
                },
                "do":[
                {
                    "event": "EventBroker.ForwardEvent",
                    "inEvent": {"event": "$Scheduler.
                        Schedule.timeoutEvent"}
                } ],
                "else": [
                {
                    "event": "Parallel.Split",
                    "do":
                    [
                        { "event": "FlowControl.Sleep", "t":
                            5000 },
                        {
```

```
                              "event": "EventBroker.
                                  ForwardEvent",
                              "inEvent":
                              {
                                  "event": "Scheduler.Schedule
                                      ",
                                  "timeoutEvent": "$Scheduler.
                                      Schedule.timeoutEvent",
                                  "timeout": "$Scheduler.
                                      Schedule.timeout",
                              }
                          }
                      ]
                  } ]
              } ]
          }
      },
      {
          "event": "BehaviourAdvertising.RegisterBehaviour",
          "behaviourDescription":
          {
              "behaviour": "Scheduler",
              "description": "Schedules a event for a future
                  time",
              "capabilities": [
              {
                  "capability": "Schedule",
                  "required": ["Math", "FlowControl"],
                  "parameters": [
                      {
                          "name": "timeoutEvent",
                          "type": "Event",
                          "required": "true"
                      },
                      {
                          "name": "timeout",
                          "type": "Timestamp",
```

```
                                "required": "true"
                            }
                        ]
                    } ]
                }
            } ]
        }
```

In this rule we create and register both *Schedule* high-level capability rule and the respective behaviour description. This is not a particularly efficient implementation of a scheduler, but more of a test of how to create a high-level capability, bundle it into a behaviour and display it on an object. This implementation of the *Schedule* high-level capability has a precision of five seconds (configurable from the *Sleep* capability's parameters), as it checks if the scheduled time has arrived with this periodicity. For this concrete behaviour (and in this scenario) it is irrelevant which object displays the behaviour therefore we did not need to specify a target object. We can consider then that this behaviour is displayed by Open Super-Object as a whole. If we want to specify exactly which object displays a new behaviour, we can specify the target of the *BehaviourAdvertising.RegisterBehaviour* event.

The procedure for creating a new behaviour is repetitive and not necessarily intuitive for an end-user, therefore we took an extra step of abstracting it by creating and registering a *Behaviour Creator* behaviour using the rule in Listing **??**.

Listing 6.8: Registering a Scheduler Behaviour

```
{
    "rule": "BehaviourCreator",
    "event": "startup",
    "actions": [
    {
        "event": "RuleRepository.RegisterRule",
        "rule": {
            "rule": "Create Behaviour"
            "event": "BehaviourCreator.CreateRule",
```

```
            "actions": [
            {
                "event": "FlowControl.ForEach",
                "list": $BehaviourCreator.CreateRule.rules,
                "element": "rule",
                "do": [
                {
                    "event": "RuleRepository.RegisterRule",
                    "rule": $rule
                } ]
            },
            {
                "event": "BehaviourAdvertising.
                    RegisterBehaviour",
                "to": $BehaviourCreator.CreateRule.object
                "behaviourDescription": $BehaviourCreator.
                    CreateRule.behaviourDescription
            } ]
        }
    },
    {
        "event": "BehaviourAdvertising.RegisterBehaviour",
        "behaviourDescription":
        {
            "behaviour": "BehaviourCreator",
            "description": "Handles the creation of a new
                Behaviour",
            "capabilities": [
            {
                "capability": "RegisterRule",
                "required": ["FlowControl"],
                "parameters": [
                    {
                        "name": "rules",
                        "type": "Rule",
                        "required": "true"
                    },
```

```
                    {
                        "name": "behaviourDescription",
                        "type": "BehaviourDescription",
                        "required": "true"
                    },
                    {
                        "name": "object",
                        "type": "URI",
                        "required": "false"
                    }
                ]
            } ]
        }
    } ]
}
```

6.7 Conclusions

In this chapter we demonstrated a few case studies where we introduced and tested different aspects and challenges of the framework. We showed how to:

- Distribute and orchestrate functionality across objects and platforms using rules;
- Control the flow of execution and perform calculations and test conditions using independent system behaviours;
- Design a rule using capability request chaining and the advantages of top-down execution;
- Prioritise one rule over another;
- Design a system aiming at resilience of services.

While maintaining an open HTTP connection when requesting a capability (or the execution of a rule) brings some advantages (simplifies implementation, for example), it also brings certain disadvantages. As observed during the experiments, certain systems may benefit from rules that are running continuously, which would eventually cause a connection timeout. In this chapter we addressed this issue by proposing the strategy of branching a rule execution so

that the main branch is terminated (closing and responding to the the initial HTTP request) while the rule may continue executing on a secondary branch.

During these experiments we developed certain reusable strategies and behaviours in order to tackle a number of different situations when implementing an Open Object System. On the next chapter we propose a set of usage patterns based on these.

7

Evaluation

This chapter outlines the current status and a critical review of the features, advantages and limitations of the framework implementation. We also compare it with other existing platforms using qualitative metrics based on the target characteristics pointed out in Chapter 2.

7.1 Status

The current implementation supports the different components of the framework described in Chapter 4 (Capabilities, Behaviours, Events and Rules) and the System Behaviours described in Chapter 5, which were tested in a series of experiments, the most meaningful being described in Chapter 6. All the system behaviours communicate with each other by exchanging events in a distributed and decentralised way and the applications are orchestrated by rules. These rules orchestrate the interactions between objects in a loose-coupled way, through the abstraction of behaviours that the objects display. The implementation supports interactions between objects with partial implementations

— objects that do not display some or all system behaviours as these can be shared / outsourced. We have developed a full implementation written in Java (for prototype development convenience) that can be run on a desktop or laptop computer, and two partial implementations, one written in C that runs on an Arduino[1] microcontroller and another written in Python that runs on a computer or on a Raspberry Pi[2].

We designed the framework with EUD in mind from various perspectives. The Event Condition Action (ECA) rule based system and the behaviour abstractions and descriptions, for example, make the framework a suitable candidate to be the target of an EUD tool such as [DDRF10], [RMMH+09] or [DP12]. It is also possible to describe some internal functionalities of the objects (namely, high level capabilities that deal with only internal API's indirectly, through the use of low level capabilities) using ECA rules that can be modified by the users in order to further adapt an object's behaviour to their personal objectives and routines (a concept described as Meta-design [FGY+04]).

In order to reduce complexity when dealing with external systems of groups of Open Objects, we abstract a group of objects that is able to work independently (displaying all system behaviours and orchestrating itself through the use of rules) as an Open Super-Object. Open Super-Objects as a whole can display their own behaviours and act as a single object to the outside. This concept makes it simpler to integrate and orchestrate clusters of objects.

A further advantage of the framework is the fact that very lightweight devices, more capable computers, and external services can coexist in the same environment and are treated equality as first-class citizens of the system. Services that objects (of any kind) provide are treated as capabilities and abstracted with behaviours. This avoids the need for object proxies; however, there is the possibility of using objects as proxies or wrappers of external services or objects that do not implement a Web interface (or implement one that is incompatible with the framework, e.g. by using a proprietary message format), and expose their methods through the use of a custom behaviour description, as explained in Chapter 4 and 6.

When dealing with ad-hoc scenarios and dynamic networks, failure (e.g. er-

[1]http://www.arduino.cc
[2]http://www.raspberrypi.org

rors, service unavailability, unstable connections) is pervasive. Failure is an integrated part of the framework. Behaviour Mapping abstracts away behaviours from objects allowing for functionality to be defined and to be used independently of the availability of objects at run-time — i.e. we separate functionality from who is providing it. Unavailability of behaviours is also handled as an integrated part of rules and is, in fact, considered not as an exception but as a normal condition of the execution — e.g. using the *onError* parameter of a capability request to prioritise one rule over another.

The current implementation was designed with interoperability in mind by offering Web interfaces to object's capabilities (and even to the system's capabilities) that are easily interfaced from other platforms, and vice-versa, by writing behaviour descriptions that treat external web services as capabilities, becoming usable from an Open Object environment. The interfaces are not, however, purely RESTful. We do not, for example, make use of all the different HTTP methods as one would in a RESTful service (see [Wil07] for instance), which would make the objects' interface more standard and therefore easier to interface with from other systems.

Usability and extensibility are important concerns of user-centric systems and in particular one that aims at *EUD*. Usability includes aspects such as the ease of writing and reusing rules, adding objects to an Open Object environment, or, from an object maker side, the simplicity of adding the necessary capabilities to an object in order to make it compatible with the framework.

It would be relatively straightforward to develop an EUD layer over the Rules and the Behaviour Descriptions as the metaphor used for the rules is simple enough to be understood by a non technical person, as suggested by [DDRF10] which uses a similar rule structure, as far as request chaining goes, although further studies would need to be carried out to support this. A possible EUD tool could, for example, read behaviour definitions and suggest capabilities to attach to capability parameters based on their type, or provide access to the different event properties when creating a rule or writing expressions in order to avoid the need to memorise them. It is also simple to progress from very simple rules to complex ones, and from there to encapsulating reusable actions and creating new capabilities, as the metaphors involved in these tasks are very similar, regardless of the complexity of the system, aiming at a smooth learning

curve.

From the point of view of object manufactures, assuming network connectivity, adapting an object (or an external Web Service, for that matter) to work within an Open Object environment can be done at different levels:

1. External service with a Behaviour Description that describes the service and how to interact with it. The Behaviour Description may be provided by a third party and must be manually registered in the Behaviour Mapping;
2. The object displays a Behaviour Advertising behaviour and is able to discover and/or be discovered by a Behaviour Mapping behaviour;
3. The object displays other or all system behaviour and is able to assist with system management tasks, such as storage/execution rules or event brokerage.

Having different levels of adaptation when augmenting an object to interact with the Open Object framework, but still being able to be operate within it without a proxy object, effectively lowers the development and cost barriers for manufacturers to add Open Object support.

Extending an object with new functionalities is also extremely simple. In order to add Parallelisation, for instance, one can use any hardware or software platform and programming language that is found suitable for the task and then exposing this feature as one or more web accessible capabilities These capabilities are then described by a behaviour description registered on a behaviour mapping, deployed locally or externally, making it integratable seamlessly with the rest of the framework.

This implementation presents, however, some limitations. We did not implement security, data privacy, or access control layers, being conscious that, although these aspects are critical to the kind of applications envisioned in our scenarios, by using Web technologies there is an extended range of tried and tested solutions to cope with these challenges. Approaches such as those shown in [TB06], [KZL+01], [HPJ05] or [CCS+11] could potentially be adapted to secure the data and the access to personal objects of the users.

Scalability has also not been the main focus of this thesis, as the scenarios we aim at are small domestic applications, simple enough to be managed by a non-

technical end-user or group of end-users. However, with the number of personal devices increasing, this issue might become increasingly significant, thus further research should study the effects of scalability of the framework. Furthermore, as the scenarios scale up, problems such as redundancy and inconsistency arise. For example, some rules might contradict or undo others, particularly in cases where several rule repositories cohabit the same OO environment or when network connection between nodes is inconsistent. These issues are left for the user to solve through carefully designing rules that check for and work around these issues as these are not addressed by the framework.

7.2 Comparative Evaluation

In this section, we evaluate the framework by having a critical discussion around a qualitative comparison of other existing platforms (of both academic and industrial nature), against features that are directly tied to the requirements proposed in Chapter 2. For evaluation purposes, we have chosen a number of IoT platforms: Contiki [DGV04], ThingSpeak[3], EVRYTHNG[4], Xively[5], ioBridge[6], HomeOS [RSL+04], Lab of Things[7]. One interesting approach for building IoT system is to look at them as resources managed by software agents in a multi-agent system. Keeping that in mind, we also selected a few multi-agent systems platforms as terms of comparison: EVATAR [DS10], Super-Agents [Sta10] and Self-Managed Cells [SBD+05]. The work on Super-Agents outlines an architectural framework for governing interactions between resources, which can be seen as a parallel of Open Objects' capabilities. The concept of Open Super-Object was taken from this work, although our approach is not compatible with the principles of agent computing, as the internal logic of an agent is traditionally immutable.

All of the platforms selected were chosen based on the variety of their nature (centralised/decentralised, academic/industrial, cloud-based/local, experimental/production), state of maturity, target audiences. We are not assessing the performance of the current implementation of the framework as we did not fo-

[3] http://www.thingspeak.com
[4] http://www.evrythng.com
[5] http://www.xively.com
[6] http://www.iobridge.com
[7] http://www.lab-of-things.com/

cus on efficiency during implementation. Several components and mechanisms could be further optimised, such as caching connectivity status of objects in the Behaviour Mapping behaviour, or caching previously used behaviour-object mapping on the Rule Execution behaviour. Also, for the purposes of the experiments, we did not implement certain trivial tasks, such as evaluation of mathematical expressions, and instead we used individual capabilities for each operation, which considerably increased the number of requests per rule.

Platforms	End-User Development	Ad-hoc interactions	Decentralised	Lightweight	Interoperability	Application Layer	Meta-Design	Security / Privacy Layer	Mature Platform
Open Objects	-	+	+	+	+	+	+		
Contiki		+	+	+	+			+	+
ThingSpeak				-	+			+	+
EVRYTHNG	-			-	+			+	+
Xively				-	+			+	+
ioBridge	-			-	+	-			+
HomeOS				-				+	-
Lab of Things				-	+				-
EVATAR				-		+			
Super-Agents						+			
Self-Managed Cells	-	+	+			+	+	+	

Table 7.1: Systematic comparison between a number of platforms according to a number of characteristics. **"+"**: supported; **"-"**: partially support; **" "**: no evidence of support.

The metrics used in the systematic comparison (Table 7.1) are derived from the requirements in Chapter 2. In respect to **End-User Development**, we are assessing if there is an EUD tool available for the platform, or how simple it would be to implement one. We consider the Open Objects platform to have partial support for EUD in the sense that the ECA rules and the behaviour are

designed to have metaphors close to one that would work on a EUD tool, as discussed previously. Rules, behaviours and capabilities can easily be translated into a block language such as [P+96], [DP12] or [DDRF10]. Some experimental studies [Gui10] were carried out in order to adapt the visual programming tool ClickScript [Nae09] to the Web of Things platform that later became EVRYTHNG. Also, ioBridge has an interface for creating simple actions (such as HTTP requests) to be triggered by events coming from devices.

While it is an inherent part of the Open Objects framework design, other platforms also provide support for **decentralised** and **ad-hoc interactions** between objects. Contiki is a lightweight operating system that is embedded on each device, enabling them to interact directly with eachother and, depending on support from the application layer, be used on ad-hoc situations. Self-Managed Cells can also engage in ad-hoc peer-to-peer interactions as shown in [SFLD+07].

In the context of this investigation, an important aspect of decentralised frameworks is being **lightweight** enough to be embedded into low-powered devices with little computation and memory capabilities (such as those based on microcontrollers), which is the case of Open Objects and Contiki. For the purposes of evaluation, we considered "partially" lightweight, frameworks that allow lightweight objects to be used directly, without the use of custom software or hardware proxies, which is the case of the cloud-based solutions (ThingSpeak, EVRYTHNG, Xively, ioBridge) but also the middleware-based ones (HomeOS/Lab of Things and EVATAR).

Being **interoperable** with other systems is key in the Internet of Things as heterogeneity of platforms and protocols is the rule. In this sense, we consider that a platform has a higher level of interoperability when it uses standard or commonly used communication protocols (e.g. providing RESTful API's) instead of closed proprietary formats, for example. Some of the platforms assessed do not include an **application layer** (Lab of Things can be considered an application layer over HomeOS), which makes it a necessity to have solid interoperability. Others, such as EVATAR, Super-Agents and Self-Managed Cells operate in a closed environment and rely on proprietary message formats, which does not mean they cannot interact with other systems, instead, these require custom software proxies to do so. In the current implementation of Open Objects, capabilities are exposed and accessible through HTTP requests,

and their API's (the Behaviour Descriptions) are open and simple to make use of externally. Additionally, external services can be accessed by encapsulating them with simple Behaviour Descriptions, although common service description languages such as RSDL [Pas12] or WSDL [CDK+02] cannot be read and used directly at the time of writing.

Another important distinction between platforms is that Open Objects, EVATAR, Super-Agents and Self-Managed Cells can work as closed and complete systems with its own application logic (in the form of rules, policies or agent minds) written into their **application layer**. The others in this analysis are base platforms that provide access to objects' resources for external usage, hence not including an application layer themselves. ioBridge, however, provides a simple rule system that could be considered support for applications based on simple interactions between objects.

We see **Meta-Design** in this context as applications built in such way that give end-users ability to change the behaviour of its applications. Self-Managed Cells, for example, have Ponder2 [TLDS08] policy-based rules that describe the behaviour of its applications. These policies can be changed during runtime, thus giving the end-user (with the aid of an appropriate policy editor) the possibility to further improve or adapt their system.

When dealing with personal artefacts, **data protection and access control** are of utmost importance. As explained before, Open Objects (alike ioBridge, EVATAR, Super-Agents and Self-Managed Cells) do not offer either security or an access control layer. As Open-Objects' communication is done through common web protocols, one can assume that typical security mechanisms can be put in place, although this requires further investigation, particularly as decentralised and ad-hoc applications present challenges related to trust, for example.

Finally, in terms of **platform maturity**, some of the platforms reviewed, namely Contiki, ThingSpeak, EVRYTHNG, Xively and ioBridge, are well established in the industry, with a large number of applications built onto them. The others have a more experimental nature, thus being confined to controlled research environments such as EVATAR, Super-Agents and Self-Managed Cells, the latter having a substantial research work [SF09, SBD+05, DLS+05, SFLD+07] put into it. Although the Open Objects framework is still in its infancy, having

some experimental prototypes, and little experimentation on real-world scenarios, our empirical experience with it shows ease of use, simplicity of implementation, and potential for becoming a mature platform once security mechanisms are implemented and an appropriate EUD tool is developed.

7.3 Summary

In Chapter 2, we discussed a number of target requirements we would be aiming at during the development of the present study. In this chapter, we presented a critical discussion of the current status of the framework. We have selected an heterogeneous range of ten other academic and industrial platforms to review comparatively. Our evaluation shows that, despite being in its infancy and lacking important features (particularly related to security), the current implementation offers a significant number of characteristics that makes it unique in the field, and appropriate to be used in the scenarios envisioned throughout this thesis.

8
Conclusions

This chapter provides final remarks with an overview of the work carried out and its contributions, and a discussion of future work potentiated by the case-studies and evaluation.

8.1 Review and Discussion of the Achievements

While we witness an increasing number of cheap small computational power that is heterogenous and more and more segmented, we wondered where this would lead us, what our world would look like in the next years. We envisioned a reality where people are given back control over their physical objects and over what they do, instead of what is dictated by those who design and produce them. We believe that in many cases — such as the examples covered in this thesis — giving people simple ways of controlling the functionality of their devices has the potential to be more powerful than designing highly intelligent systems that try to predict people's activities and desires. After all, No one enjoys having their work interrupted by a talking paper clip that "intelligently" assumed we

were doing something else. Instead, by creating tools that make it simple for users to control and define actions that occur with the objects that surround them, these actions should become easier to predict and in case of errors, the user has more control over the system and should be more able to understand and fix the problem. Naturally, these are our speculations and need further study.

End-User Development (EUD) is defined as a field that studies metaphors and techniques that empower users by enabling them to develop and modify the tools used in their everyday lives. With this objective in mind, we set off to design a framework to support EUD over computationally capable physical objects and services provided by external sources (such as weather forecasts or cloud-based file storage). These scenarios represent highly dynamic systems composed of heterogeneous devices with limited capabilities. User-oriented orchestration of functionality in these kinds of systems presents a number of challenges. Each of these challenges and our approach to them is briefly discussed below.

In order to provide support for EUD, we define the interactions between objects as rules that can be added and modified through the lifetime of the system. In our implementation, these rules are specified using a custom syntax that is easily mappable into a thin EUD layer, and expressive enough to describe complex functionalities orchestrating a distributed and decentralised system composed of domain dependent, domain independent and system behaviours. Complex matters such as decentralisation, distribution, message passing and behaviour to object mapping are abstracted away from the user and handled by System Behaviours. An important aspect is that in order for the rules to make sense to the common user, capabilities and behaviours need to be described in a simple and clear way.

The concept of Behaviour not only simplifies and standardises the usage of objects, but also focuses the rule creation on functionality rather than on the objects themselves. This facilitates the creation of rules that produce ad-hoc and/or resilient interactions. Furthermore, the fact that rules can be used to define new capabilities and add new behaviours (or modify existing) to existing objects, allows for the Meta-Design of objects. We further pushed the concept of Meta-Design by introducing the concept of Hollow Object: an Open Object that possesses low-level capabilities to interact with its sensors and actuators,

but is not designed with any particular functionality in mind. Instead, it is provided as a generic building block for the user to add functionality to, or to be used as part of an application composed of a group of objects.

In order to achieve a high level of decentralisation while keeping all objects treated equally as first-class citizens within the framework, we had to develop an implementation that was lightweight enough to be installed on very hardware constrained devices such as small microcontrollers. Our approach was to consider that all components of the framework (including system management ones) should be split into atomic, independent and distributable behaviours without a shared memory pool. We also approached the implementation with sharing and outsourcing of capabilities in mind. This means that (1) an object is able to share the capabilities it possesses with other objects and (2) an object can outsource a particular capability that it does not possess to another object. This makes it possible to have partial implementations of the framework on more lightweight objects, as long as, as a whole, the group of objects plus any external services involved in an application can display all the System Behaviours.

In today's world of interconnected services and omnipresent information, interoperability of systems is crucial. In our implementation, we have approached interoperability at every level of the framework. Behaviours (and their capabilities) can be accessed and requested through standard Web interfaces described by behaviour descriptions, which are defined in an open JSON format, simple to interpret by external systems. All system components are themselves behaviours, therefore, easily accessible. Furthermore, because system behaviours are atomic and independent, one can imagine applications where some or all of them are effortlessly replaced by custom alternatives that comply with the same behaviour descriptions, e.g. a more advanced event broker that synchronises events with other event brokers. In the current implementation, rules are also specified in JSON and can be created or modified from outside the system. Different rule creation tools can be developed in order to target applications with specific needs, for example.

8.2 Future Work

This thesis lays the ground for some future work, described ahead.

8.2.1 End-User Development Tool

The aim of this thesis was to produce a framework and an implementation that offers support for End-User Development. Therefore, the main path for future investigation is towards adapting or creating a tool that makes it easy enough for a common user to define rules and create new capabilities and behaviours. The rules are designed to map directly into a data-flow block based visual programming language (such as [DDRF10, DP12]), although other kinds of tools could also be adapted to comply with the current rule specification.

8.2.2 Maturing the Platform

In Chapter 7 we outlined some limitations of the current implementation of the framework. These will naturally be the next steps of development. There needs to be further research into the issues of Privacy and Trust on decentralised and unstable networks of objects controlled by end-users. This is key when maturing the framework as in our scenarios we envision objects that have access to sensitive and private data. Also, it is important to address problems of redundancy and inconsistency of rules at the framework level, to allow it to be used on larger scale and more complex scenarios. Furthermore, there needs to be future practical work on implementing or adapting an existing efficient mathematical expression engine and methods for validating rules and behaviour descriptions.

8.2.3 On Predicates First vs Context First when designing an End-User Development language

As a side conclusion, during our work and research on the rule language we saw a pattern emerging: when people communicate, they often provide a subject and a predicate first, and then narrow down the context. For instance, in the phrase “Search for The Beatles on my phone and play it on my Hi-Fi”, the context is provided after each predicate. When an instruction is provided in such way, it’s harder for a machine to assist the user during its formulation compared to an instruction of the kind “Hi-Fi play phone music search The Beatles”. The reason is that when we provide context first, the number of options is smaller: the Hi-Fi has a relatively small number of actions that it can perform (e.g. “play”, “stop”, “pause”, etc); the action “play” is expecting music, therefore

the machine may choose to present the user options of this type next; the action "search" expects the user to type in search terms related to music (e.g. genres, authors, etc), therefore a suitable interface may be presented to the user. This is one of the differences between machine languages and human languages and it is also made obvious in object-oriented languages: we would type the sequence of the kind "if car has petrol then turn car on" but we may think "turn on the car if it has petrol". As opposed to when a software developer provides instructions to a machine, a normal user is not expected to adapt its language to the machine language: systems should try to bridge the gap between its language and the one of the user, not the other way round. We believe this is an area that requires some further studying as it seems to be a fundamental gap in the human-machine communication bridge.

Bibliography

[ADLH$^+$02] Bob Atkinson, Giovanni Della-Libera, Satoshi Hada, Maryann Hondo, Phillip Hallam-Baker, Johannes Klein, Brian LaMacchia, Paul Leach, John Manferdelli, Hiroshi Maruyama, et al. Web services security (ws-security). *Specification, Microsoft Corporation*, 2002.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[AM07] Juan Carlos Augusto and Paul McCullagh. Ambient intelligence: Concepts and applications. *Computer Science and Information Systems/ComSIS*, 4(1):1–26, 2007.

[AWD04] Mehran Abolhasan, Tadeusz Wysocki, and Eryk Dutkiewicz. A review of routing protocols for mobile ad hoc networks. *Ad hoc networks*, 2(1):1–22, 2004.

[BHBR10] Carl Bergenhem, Qihui Huang, Ahmed Benmimoun, and Tom Robinson. Challenges of platooning on public motorways. In *17th World Congress on Intelligent Transport Systems*, pages 1–12, 2010.

[BLFF96] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext transfer protocol–http/1.0, 1996.

[CAJ09] Diane J Cook, Juan C Augusto, and Vikramaditya R Jakkula. Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, 5(4):277–298, 2009.

[CCS+11] Dong Chen, Guiran Chang, Dawei Sun, Jiajia Li, Jie Jia, and Xingwei Wang. Trm-iot: A trust management model based on fuzzy reputation for internet of things. *Computer Science and Information Systems*, 8(4):1207–1228, 2011.

[CDK+02] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet computing*, 6(2):86–93, 2002.

[CFMD10] Joëlle Coutaz, Emeric Fontaine, Nadine Mandran, and Alexandre Demeure. About composing our own smart home. In *AVI*, pages 405–406, 2010.

[CK03] Chee-Yee Chong and Srikanta P Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003.

[CoA12] Application Protocol CoAP. Coap: An application protocol for billions of tiny internet nodes. 2012.

[Cos07] Roger L Costello. Building web services the rest way. *UR L: http://www. xfront. com/REST-Web-Services. html. Ultima Consulta*, 11:2007, 2007.

[CSDC11] Walter Colitti, Kris Steenhaut, and Niccolò De Caro. Integrating wireless sensor networks with the web. *Extending the Internet to Low power and Lossy Networks (IP+ SN 2011)*, 2011.

[DBS+01] Ken Ducatel, Marc Bogdanowicz, Fabiana Scapolo, Jos Leijten, and Jean-Claude Burgelman. *Scenarios for ambient intelligence in 2010*. Office for official publications of the European Communities, 2001.

[DDRF10] José Danado, Marcin Davies, Paulo Ricca, and Anna Fensel. An authoring tool for user generated mobile services. In *Future Internet-FIS 2010*, pages 118–127. Springer, 2010.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[DHC05] Faiyaz Doctor, Hani Hagras, and Victor Callaghan. A fuzzy embedded agent-based approach for realizing ambient intelligence in intelligent inhabited environments. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 35(1):55–65, 2005.

[DLS+05] Naranker Dulay, Emil Lupu, Morris Sloman, Joe Sventek, Nagwa Badr, and Stephen Heeps. Self-managed cells for ubiquitous systems. In *Computer Network Security*, pages 1–6. Springer, 2005.

[DP12] José Danado and Fabio Paternò. Puzzle: a visual-based environment for end user development in touch-based mobile phones. In *Human-Centered Software Engineering*, pages 199–216. Springer, 2012.

[DS10] Nikolaos Dipsis and Kostas Stathis. Evatar–a prototyping middleware embodying virtual agents to autonomous robots. In *Ambient Intelligence and Future Trends-International Symposium on Ambient Intelligence (ISAmI 2010)*, pages 167–175. Springer, 2010.

[EFGK03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[Ehn08] Pelle Ehn. Participation in design things. In *Proceedings of the Tenth Anniversary Conference on Participatory Design 2008*, pages 92–101. Indiana University, 2008.

[Erl08] Thomas Erl. *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008.

[Fer99] Jacques Ferber. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.

[FG06] Gerhard Fischer and Elisa Giaccardi. Meta-design: A framework for the future of end-user development. In *End user development*, pages 427–457. Springer, 2006.

[FGM+99] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[FGY+04] Gerhard Fischer, Elisa Giaccardi, Yunwen Ye, Alistair G Sutcliffe, and Nikolay Mehandjiev. Meta-design: a manifesto for end-user development. *Communications of the ACM*, 47(9):33–37, 2004.

[FS11] Kevin R Fall and W Richard Stevens. *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[FW99] Klaus Finkenzeller and Rachel Waddington. *RFID handbook: radio-frequency identification fundamentals and applications*. Wiley New York, 1999.

[Gar05] Lars Marius Garshol. Bnf and ebnf: What are they and how do they work. 16, 2005.

[GBCM11] Frieder Ganz, Payam Barnaghi, Francois Carrez, and Klaus Moessner. Context-aware management for sensor networks. In *Proceedings of the 5th International Conference on Communication System Software and Middleware*, page 6. ACM, 2011.

[GHL05] Alfonso Gárate, Nati Herrasti, and Antonio López. Genio: an ambient intelligence application in home automation and entertainment environment. In *Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies*, pages 241–245. ACM, 2005.

[Gib10] Alicia M Gibb. *New media art, design, and the Arduino microcontroller: A malleable tool*. PhD thesis, Pratt Institute, 2010.

[GT09] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*

(MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain, page 15, 2009.

[GTW10] Dominique Guinard, Vlad Trifa, and Erik Wilde. A resource oriented architecture for the web of things. In *Internet of Things (IOT), 2010*, pages 1–8. IEEE, 2010.

[Gui10] Dominique Guinard. *Mashing up your web-enabled home*. Springer, 2010.

[Hal84] Daniel Conrad Halbert. *Programming by example*. PhD thesis, University of California, Berkeley, 1984.

[HDK08] Bjorn Hartmann, Scott Doorley, and Scott R Klemmer. Hacking, mashing, gluing: Understanding opportunistic design. *Pervasive Computing, IEEE*, 7(3):46–54, 2008.

[Hil92] Daniel D Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[HKS09] Stephan Haller, Stamatis Karnouskos, and Christoph Schroth. *The internet of things in an enterprise context*. Springer, 2009.

[HMF+08] Atsushi Hashimoto, Naoyuki Mori, Takuya Funatomi, Yoko Yamakata, Koh Kakusho, and Michihiko Minoh. Smart kitchen: A user centric cooking support system. In *Proceedings of IPMU*, volume 8, pages 848–854, 2008.

[HPJ05] Yih-Chun Hu, Adrian Perrig, and David B Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. *Wireless networks*, 11(1-2):21–38, 2005.

[IGH+11] Valérie Issarny, Nikolaos Georgantas, Sara Hachem, Apostolos Zarras, Panos Vassiliadist, Marco Autili, Marco Aurélio Gerosa, and Amira Ben 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.

[KEW02] Bhaskar Krishnamachari, Deborah Estrin, and Stephen Wicker. The impact of data aggregation in wireless sensor networks. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 575–578. IEEE, 2002.

[KKFS10] Gerd Kortuem, Fahim Kawsar, Daniel Fitton, and Vasughi Sundramoorthy. Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51, 2010.

[KT07] Hiroko Kato and Keng T Tan. Pervasive 2d barcodes for camera phone applications. *Pervasive Computing, IEEE*, 6(4):76–85, 2007.

[KZL+01] Jiejun Kong, Petros Zerfos, Haiyun Luo, Songwu Lu, and Lixia Zhang. Providing robust and ubiquitous security support for mobile ad hoc networks. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 0251–0251. IEEE Computer Society, 2001.

[Law08] George Lawton. Developing software online with platform-as-a-service technology. *Computer*, 41(6):13–15, 2008.

[LPKW06] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-user development: An emerging paradigm*. Springer, 2006.

[MB76] Robert M Metcalfe and David R Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[MD88] Paul Mockapetris and Kevin J Dunlap. *Development of the domain name system*, volume 18. ACM, 1988.

[MF10] Friedemann Mattern and Christian Floerkemeier. From the internet of computers to the internet of things. In *From active data management to event-based systems and more*, pages 242–259. Springer, 2010.

[Mul07] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82. ACM, 2007.

[Mye90] Brad A Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990.

[Nae09] Lukas Naef. *ClickScript a visual programming language in the browser*. PhD thesis, Master thesis, ETH Zurich, Switzerland, 2009.

[NL94] Jakob Nielsen and Jonathan Levy. Measuring usability: preference vs. performance. *Communications of the ACM*, 37(4):66–75, 1994.

[NRL07] Pavel V Nikitin, KVS Rao, and Steve Lazar. An overview of near field uhf rfid. In *IEEE international Conference on RFID*, volume 167. Citeseer, 2007.

[P+96] Miller Puckette et al. Pure data: another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[P+99] Bruce Perens et al. The open source definition. *Open sources: voices from the open source revolution*, pages 171–85, 1999.

[Pas12] Michael Pasternak. Restful service description language, October 19 2012. US Patent App. 13/656,032.

[Per08] Charles E Perkins. *Ad hoc networking*. Addison-Wesley Professional, 2008.

[PPW06] Alexandra Poulovassilis, George Papamarkos, and Peter T Wood. Event-condition-action rule languages for the semantic web. In *Current Trends in Database Technology–EDBT 2006*, pages 855–864. Springer, 2006.

[QBVG08] Till Quack, Herbert Bay, and Luc Van Gool. Object recognition for the internet of things. In *The Internet of Things*, pages 230–246. Springer, 2008.

[R+99] Eric S Raymond et al. The magic cauldron, 1999.

[Res01] Eric Rescorla. *SSL and TLS: designing and building secure systems*, volume 1. Addison-Wesley Reading, 2001.

[RMMH+09] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[RSL+04] Neal Rosen, Rizwan Sattar, Robert W Lindeman, Rahul Simha, and Bhagirath Narahari. Homeos: Context-aware home connectivity. In *International Conference on Wireless Networks*, pages 739–744, 2004.

[RSP11] Paulo Ricca, Kostas Stathis, and Nick Peach. A lightweight service registry for unstable ad-hoc networks. In *Ambient Intelligence*, pages 136–140. Springer, 2011.

[SBD+05] Joseph Sventek, Nagwa Badr, Naranker Dulay, Steven Heeps, Emil Lupu, and Morris Sloman. Self-managed cells and their federation. 2005.

[SF09] Alberto E Schaeffer Filho. *Supporting management interaction and composition of self-managed cells.* PhD thesis, Imperial College London, 2009.

[SFLD+07] Alberto Schaeffer-Filho, Emil Lupu, Naranker Dulay, Sye Loong Keoh, Kevin Twidle, Morris Sloman, Steven Heeps, Stephen Strowes, and Joe Sventek. Towards supporting interactions between self-managed cells. In *Self-Adaptive and Self-Organizing Systems, 2007. SASO'07. First International Conference on*, pages 224–236. IEEE, 2007.

[SGAP00] Katayoun Sohrabi, Jay Gao, Vishal Ailawadhi, and Gregory J Pottie. Protocols for self-organization of a wireless sensor network. *IEEE personal communications*, 7(5):16–27, 2000.

[SK11] Subashini Subashini and V Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1–11, 2011.

[SKG+09] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, LMSD Souza, and Vlad Trifa. Soa-based integration of the internet of things in enterprise services. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 968–975. IEEE, 2009.

[SKPB07] Kostas Stathis, Stella Kafetzoglou, Symeon Papavassiliou, and Stefano Bromuri. Sensor network grids: Agent environments combined with qos in wireless sensor networks. In *Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on*, pages 47–47. IEEE, 2007.

[SS09] Fariba Sadri and Kostas Stathis. Ambient intelligence. *Ubiquitous and Pervasive Computing: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 121, 2009.

[Sta10] Kostas Stathis. Autonomic computing with self-governed superagents. In *Self-Adaptive and Self-Organizing Systems Workshop (SASOW), 2010 Fourth IEEE International Conference on*, pages 76–79. IEEE, 2010.

[STF07] Dairazalia Sanchez, Monica Tentori, and Jesus Favela. Hidden markov models for activity recognition in ambient intelligence environments. In *Current Trends in Computer Science, 2007. ENC 2007. Eighth Mexican International Conference on*, pages 33–40. IEEE, 2007.

[TB06] George Theodorakopoulos and John S Baras. On trust models and trust evaluation metrics for ad hoc networks. *Selected Areas in Communications, IEEE Journal on*, 24(2):318–328, 2006.

[TLDS08] Kevin Twidle, Emil Lupu, Naranker Dulay, and Morris Sloman. Ponder2-a policy environment for autonomous pervasive systems.

In *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on*, pages 245–246. IEEE, 2008.

[TP06] Scott E Thompson and Srivatsan Parthasarathy. Moore's law: the future of si microelectronics. *Materials Today*, 9(6):20–25, 2006.

[Tri11] Vlad Mihai Trifa. *Building blocks for a participatory Web of things*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 19890, 2011, 2011.

[Wan06] Roy Want. An introduction to rfid technology. *Pervasive Computing, IEEE*, 5(1):25–33, 2006.

[Wan11] Guanhua Wang. Improving data transmission in web applications via the translation between xml and json. In *Communications and Mobile Computing (CMC), 2011 Third International Conference on*, pages 182–185. IEEE, 2011.

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.

[Wil07] Erik Wilde. Putting things to rest. *School of Information*, 2007.

[YWZ+04] Kun Yue, Xiao-Ling Wang, Ao-Ying Zhou, et al. Underlying techniques for web services: A survey. *Journal of software*, 15(3):428–442, 2004.

[ZEBD98] Eli Zelkha, Brian Epstein, S Birrell, and C Dodsworth. From devices to ambient intelligence. In *Digital living room conference*, volume 6, 1998.

Appendices

A

Open Object System Behaviours: Capability Algorithms

This Appendix lists the full pseudo-code algorithms of the System Behaviours capabilities, as explained in Chapter 5.

A.1 Behaviour Mapping

Procedure A.1 Behaviour Mapping — Register Behaviour

```
Input: BehaviourID; ObjectURI; SubscribeToEvents

if BehaviourDatabase does not contain BehaviourID then
    BehaviourDatabase .AddToTop(BehaviourID, new List)
end if
BehaviourDatabase .Get(BehaviourID) .Add(new Tuple(ObjectURI, true))
ActivateBehaviour(BehaviourID, ObjectURI)
if SubscribeToEvents then
    EventBroker .Subscribe( BehaviourID + ".*", ObjectURI)
end if
```

Procedure A.2 Behaviour Mapping — Deregister Behaviour

```
Input: BehaviourID; ObjectURI; UnsubscribeToEvents

if BehaviourDatabase has BehaviourID then
    BehaviourDatabase .Get(BehaviourID) .Remove(ObjectURI)
    Raise Event( BehaviourDeactivated( BehaviourID, ObjectURI ) )
    if SubscribeToEvents then
        EventBroker .Unsubscribe( BehaviourID + ".*", ObjectURI)
    end if
end if
```

Procedure A.3 Behaviour Mapping — Activate Behaviour

```
Input: BehaviourID; ObjectURI

if BehaviourDatabase has BehaviourID then
    BehaviourDatabase .Get(BehaviourID) .SetActive(ObjectURI, true)
end if
```

Procedure A.4 Behaviour Mapping — Deactivate Behaviour

Input: BehaviourID; ObjectURI

if BehaviourDatabase has BehaviourID **then**
    BehaviourDatabase .Get(BehaviourID) .SetActive(ObjectURI, false)
end if

Procedure A.5 Behaviour Mapping — Rank

Input: BehaviourID; ObjectURI; Rank

if BehaviourDatabase has BehaviourID **then**
    BehaviourDatabase .Get(BehaviourID) .SetRank(ObjectURI, Rank)
end if

Procedure A.6 Behaviour Mapping — Get Behaviour URIs

Input: BehaviourID
Output: ObjectURIList

for all Object in BehaviourDatabase .Get(BehaviourID) sorted by descending Ranking **do**
    ObjectURIList ← new List
    if Object .Uri is not accessible **then**
        if object .Uri is not Object .ManualRanking **then**
            Penalise(Object)
        end if
    else
        ObjectURIList .Add(Object .Uri)
    end if
    return ObjectURIList
end for

A.2 Rule Execution

Procedure A.7 Rule Execution — Execute

```
Input: Code
Output: Response

for all Instruction in Code do
    if Instruction .Execution is Bottom-up then
        for all Parameter in Instruction .Parameters do
            if Parameter .Value is Instruction then   ▷ execute instructions in parameters recursively
                Parameter .SetValue( Execute(Parameter .value))
            end if
        end for
    end if                                            ▷ simplified capability request.
    return Response ← Instruction .Request .Behaviour .Capability   ▷ simplified request capability in the parameter
end for
return Response
```

A.3 Rule Repository

Procedure A.8 Rule Repository — Register Rule

```
Input: Rule

if Rules .Get(Rule.event) is null then
    Rules .Add(Rule.event, new Map)
end if
Rules .Get(Rule.Event) .add(Rule.ID, Rule)
```

Procedure A.9 Rule Repository — Deregister Rule

```
Input: RuleID

for all Event in Rules do
    event .Remove(RuleID)
end for
```

Procedure A.10 Rule Repository — Activate Rule

```
Input: RuleID

for all Event in Rules do
    Event .Get(RuleID) .Active ← true
end for
```

Procedure A.11 Rule Repository — Deactivate Rule

```
Input: RuleID

for all Event in Rules do
    Event .Get(RuleID) .active ← false
end for
```

Procedure A.12 Rule Repository — Get Rules By Event

```
Input: Event
Output: EventRules

EventRules ← Rules .Get(Event.Type)
for all Rule in EventRules do
    if not Rule .Active or ParseCondition(rule .Condition, Event) is false then
        EventRules .Remove(Rule)
    end if
end for
return EventRules
```

Procedure A.13 Rule Repository — Get Rule

Input: RuleID

Output: Rule

for all RulesPerEvent in Rules **do**
    if RulesPerEvent contains RuleID **then**
        return RulesPerEvent .Get(RuleID)
    end if
end for
return EventRules

A.4 Event Broker

Procedure A.14 Event Broker — Forward Event

Input: InEvent
Output: Outcome
▷ Store the event in EventHistory (maintain max # of stored events)
if EventHistory .Length > MaximumHistoryLength **then**
    EventHistory .Remove(0)
end if
EventHistory .Add(InEvent)
▷ Send event to the recipients
for all Recipient in inEvent .To **do** ▷ External recipients may require parameters to be inserted into the recipient URI
    Replace Parameter Values in Recipient
    Send InEvent to Recipient
end for
▷ Look for subscribers of this event and forward it to them
if InEvent .Visibility is Public **then**
    Subscribers ← Subscriptions .GetMatches(InEvent)
    for all Subscriber in Subscribers **do**
        Send InEvent to Subscriber
    end for
end if
Rules ← RuleStorage .GetRulesByEventType(InEvent)
▷ Request Execution of Rules
for all Rule in Rules **do**
    Outcome ← RuleExecution .Execute(Rule)
end for
return Outcome

Procedure A.15 Event Broker — Subscribe

Input: EventSelector, Subscriber

```
if Subscriptions. Get(EventSelector) is null then
    Subscriptions. Add(EventSelector, new Map)
end if
Subscriptions. Get(EventSelector) .Add(Subscriber)
```

Procedure A.16 Event Broker — Unsubscribe

Input: EventSelector, Subscriber

```
Subscriptions .Get(EventSelector) .Remove(Subscriber)
```

A.5 Behaviour Advertising

Procedure A.17 Behaviour Advertising — Register Behaviour URI

Input: BehaviourDescriptionURI

```
BehaviourDescriptions .Add(BehaviourDescriptionURI)
Raise Event NewBehaviour( BehaviourDescription .BehaviourID )
```

Procedure A.18 Behaviour Advertising — Register Behaviour

Input: BehaviourDescription

```
Store BehaviourDescription into InternalURI
BehaviourDescriptions .Add(InternalURI)
Raise Event NewBehaviour(BehaviourDescription .BehaviourID )
```

Procedure A.19 Behaviour Advertising — Advertise

```
return Behaviours
```