ALMA MATER STUDIORUM — UNIVERSITY OF BOLOGNA

DISI - Department of Computer Science & Engineering
Ph.D. in Computer Science & Engineering

Ciclo XXX
Settore Concorsuale: 09/H1
Settore Disciplinare: ING-INF/05

MICRO-INTELLIGENCE FOR THE IoT: LOGIC-BASED MODELS AND TECHNOLOGIES

Candidato

ROBERTA CALEGARI

Coordinatore Dottorato

Chiar.mo Prof. PAOLO CIACCIA

Supervisore

Chiar.mo Prof. ENRICO DENTI

Co-Supervisore

Chiar.mo Prof. ANDREA OMICINI

FINAL EXAMINATION YEAR 2018

To my beloved children, Sofia, Serena, Alessandro

Acknowledgements

This thesis belongs not only to me, but also to all the researchers and professionals I had the pleasure to work with during this three-year effort.

I would like to express my sincere gratitude to my supervisor Prof. Enrico Denti for his continuous advice, encouragement, feedback and cooperation. This thesis would not be completed without his support in many regards. I wish to thank Prof. Andrea Omicini for his co-supervision during my PhD: he has been a constant and endless source of invaluably precious suggestions, criticism, and inspiration, as well as a wonderful person to share random thoughts with. A warm thanks also goes to Prof. Agostino Dovier for his invaluable collaboration. Thank you for the great help and the patience in correct my demonstrations. A special mention goes to Prof. Cecilia Mascolo and Prof. Viviana Mascardi, the reviews whose comments helped in shaping the final version of this thesis, and to Prof. Paolo Ciaccia, Prof. Stefano Rizzi, which evaluated progress of the PhD research efforts. I wish to thank Daniela Loreti, Allegra De Filippo, Ambra Molesini and all the guys of the LIA Lab and of the Apice group for the wonderful time we have had together, and for all the discussions we shared. A special mention goes to Stefano Mariani for all his help and patience in shaping the paper manuscripts and the system prototypes.

I wish to thank all the brilliant students I had the pleasure to supervise for their thesis and graduate projects, from whom I have learnt as much as I have taught.

Last, but not least, I owe everything I have to my family: my husband Simone who has constantly supported and helped me in every single important decision, my daughters Sofia and Serena who have been fundamental to refresh my energies when the work to do was just too much, and our baby Alessandro, that despite the sleepless nights, always gives a smile.

Roberta Calegari, March 8, 2018

Contents

Abstract

Computing is moving towards pervasive, ubiquitous environments in which devices, software agents and services are all expected to seamlessly integrate and cooperate in support of human objectives – anticipating needs, negotiating for services, acting on our behalf, and delivering services in an anywhere any time fashion.

An important next step for pervasive computing is the integration of intelligent agents that employ knowledge and reasoning to understand the local context and share this information in support of intelligent applications and interfaces. Such scenarios, characterised by "computation is everywhere around us", require on the one hand software components with intelligent behaviour in terms of objectives and context, and on the other their integration so as to produce *social intelligence*.

Since its inception, Logic Programming (LP) has been recognised as a natural paradigm for addressing the needs of distributed intelligence. Yet, the development of novel architectures, in particular in the context Internet of Things (IoT), and the emergence of new domains and potential applications, are creating new research opportunities where LP could be exploited, when suitably coupled with agent technologies and methods so that it can fully develop its potential in the new context. In particular, the LP and its extensions can act as *micro-intelligence sources* for the IoT world, both at the individual and the social level, provided that they are reconsidered in a renewed architectural vision. Such micro-intelligence sources could deal with the local knowledge of the devices taking into account the domain specificity of each environment.

The goal of this thesis is to re-contextualise LP and its extensions in these new domains as a source of micro-intelligence for the IoT world, envisioning a large number of small computational units distributed and situated in the environment, thus promoting the local exploitation of symbolic languages with inference capabilities. The topic is explored in depth and the effectiveness of novel LP models and architectures –and of the corresponding technology– expressing the concept of micro-intelligence is tested. In particular, two different, integrated models are presented, namely *Logic Programming as a Service* (LPaaS) and *Labelled Variables in Logic Programming* (LVLP) designed so as to act synergistically in order to support the distribution of intelligence in pervasive systems.

Chapter 1

Introduction

Pervasive computing has evolved substantially since its inception, scaling up to today's societal-scale applications such as mobile crowdsensing [JXJ$^+$15] and the *Internet of Things* (IoT) [GBMP13, AIM10, FGRS14], from smart appliances and smart environments in the early days to systems of much broader scope such as smart cities and smart transportation.

Pervasive computing calls for ubiquitous intelligence where everyday physical objects should be able to network in the IoT: intelligence is everywhere, software components are required to behave intelligently, understanding their own goals and the context where they operate; devices are required to understand each other, learn and understand situations, and understand us [LMMZ17]; in short, our everyday objects should be(come) intelligent objects in the Internet of Intelligent Things (IoIT) [ASF$^+$14]. Moreover, their integration is supposed to add social intelligence, possibly through coordination [Cas98].

At the same time, the peculiarities of pervasive contexts –such as *size* of the amount of data, information, and knowledge to handle, supporting *adaptation* and *self- management*, providing intelligence in a light-weight, *easy to use* and *customise*, *highly-interoperable* way —make engineering effective *distributed situated intelligence* far from trivial.

Logic Programming (LP) [Llo12] languages and technologies represent in principle a natural candidate for injecting intelligence within computational systems [Bro11]: yet, many issues have to be addressed—among these, the computational costs, the machinery often not suited for programming in the large (the intrinsic modularity provided by predicates does not scale up effectively, and modules are not enough for the purpose), the "no types" approach that makes it difficult to deal with domain-specific applications, the distance from mainstream programming paradigms, the integration with mainstream technologies.

Since the IoT inherently calls for a fully distributed architecture, the relationship between "LP & distribution" needs to be addressed and investigated in depth. LP seems not to fit well for nowadays perception of distributed systems, at least according to their historical meanings, since nowadays pervasive system are far away from the "simple" parallel computing of LP, arising new issues such as, among others, the universal notion of consistency of logic theory that does not cope well with the incompleteness and inconsistency intrinsically implied by a distributed scenario.

For these reasons, this thesis approaches the matter of engineering pervasive systems from the *intelligence perspective*, dealing with the aforementioned issues at the infrastructural level and re-interpreting the classical LP notions in the distributed, pervasive contexts. The aim is to exploit the potential of LP and its extensions as sources of *micro-intelligence for IoT* scenarios, in particular when coupled with agent-based technologies and methods, at both the individual and the collective level, along with an overall architectural view of IoT systems exploiting logic-based technologies.

Based on previous research efforts, we identify some major elements that need to be considered throughout the development phases of any IoT system [SGFW10]—namely, *intelligence*, *complexity*, *size*, *time & space*, and *architecture & Everything-as-a-Service*.

Here we just sketch the discussion of these features, leaving an in-depth analysis to Section 1.1.

Intelligence is intended here in the Gottfredson definition as “a very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience” [Got97]. In fact, IoT applications typically collect lots of raw data from several sources (e.g., GPS data of vehicles, real-time traffic data of road cameras, weather data) from environment sensors, as well as user-generated contents (e.g., tweets, micro-blog, check-ins, photos) from mobile social APPs, that all need to be transformed into suitable knowledge (high-level information) to be later exploited by the *collective* and *social* intelligence. In particular, sensor data have specific characteristics that need to be taken into account when reasoning and to transform them into proper knowledge—namely, they can be multi-source, heterogeneous, large-scale, continuously produced in a real-time streaming, ever-expanding, and situated both in space and time.

Complexity The IoT includes a large number of sensors, actuators and other devices that interact autonomously. Devices may appear, communicate and disappear. Interactions may differ significantly depending on the objects capabilities —some objects may have very few capabilities, limited storage and no processing capabilities at all.

Size The number of devices, and therefore of interactions, also increases significantly: billions of parallel and simultaneous events have to be managed.

Time & Space Due to the massive number of interactions, the IoT calls for proper handling of billions of parallel and simultaneous events. Real-time data processing is also essential in a wide variety of technologies, ranging from feedback control systems in vehicles and power plants to real-time imaging for minimally invasive medical procedures. Moreover, the space location, in terms on the environment in which the device is situated, plays a significant role to reason over the data and interactions are highly dependent on their locations, surroundings, and presence of other entities (e.g. objects and people) [BPRD08].

Architecture & Everything-as-a-Service Everything as a service (XaaS) [BFB$^+$11] is a category of models introduced in cloud computing [PRJ12] aiming at offering infrastructure, platforms and software applications as services —according to a more Service-Oriented Architecture (SOA) interpretation of the term "service". The XaaS model is highly efficient, scalable, and easy to use and it uses the same infrastructure IoT envisions [PZCG14]. Therefore, most of the technological solutions that are developed following the XaaS architecture, can be used to realise the vision of IoT, enabling people to benefit from ubiquitous information access.

Based on these considerations, we define the concept of *micro-intelligence* in the IoT era (Section 1.1) and explore how suitable LP models and technologies can support and promote its development. In fact, micro-intelligence could be encapsulated in devices of any sort, making them work together in groups, aggregates, societies by promoting *observability*, *malleability*, *understandability*, *formalisability*, and *norm compliance.*

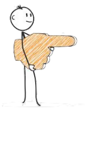
The goal of this work is both to explore the topic in depth and to develop the corresponding technology in terms of service-oriented architectures, according to the *Logic Programming as a Service* (LPaaS henceforth) approach.This approach exploits the XaaS metaphor to promote maximum availability and interoperability: any resource of any sort should be accessible as a service (possibly an intelligent one) via standard network operations.

Another key issue in the IoT context is the need to enable diverse computational models, each tailored to a specific situated component, to coherently and fruitfully coexist and cooperate within the same (logic-based) framework, so as to cope with *domain-specific* aspects. This is what the *Labelled Variables in Logic Programming* (LVLP) model is for: the added value of such a hybrid approach is to make it possible to exploit LP for what it is most suited for, such as symbolic computation, delegating other aspects – such as situated computations – to other languages or other computational levels.

1.1 Micro-Intelligence for the IoT

Moving from the Gottfredson definition of intelligence [Got97], we define *micro-intelligence* as:

MICRO
INTELLIGENCE



the capability of micro entities to reason, plan, solve problems, abstract, and learn from experience in relation to the surrounding environment.

The fundamental traits of micro-intelligence are precisely:

- the micro-level feature, influenced by the *Things* vision in IoT
- the ability to abstract, reason, plan, solve, and learn capabilities
- the situatedness feature, as the capability to interact with and act on the environment.

Micro-level feature Everyday 'things' are beginning to mix with digital systems. What used to be ordinary objects of day-to-day life, like clothing, furniture, tools, and toys, are becoming extraordinary things, seamlessly woven into global networks, infused and animated by sophisticated digital systems. PCs as we know them will disappear relatively quickly. They will give way to vast amounts of sophisticated computing and communication in our surroundings, known as *Things that Think* [HPT97, FN08].

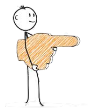
New paradigms for the Internet of Things are therefore crucial for migrating from nowadays sensor networks into networks of intelligent sensors enabled with actuation mechanisms. Such networks consists of the "Internet of Intelligent Things" (IoIT). This paradigm is the next step in the evolution of networking, for creating the experience of an ubiquitous [ASF^{+}14], and intelligent, living, internet.

One one hand, the IoT is an enabling paradigm for other forms of networking and computing, such as IoIT and Robotics as a Service (RaaS) paradigms [CDGA10]. These new paradigms propose to add intelligence to the things that are connected to the Internet, or consider things as robots that are available as a service to the users, respectively. In these scenarios, connected intelligent things are capable to solve collaboratively complex problems autonomously: by connecting and sharing ideas, large numbers of people and/or machines can provide more accurate answers to complex problems than single individuals [AIMN12].

On the other hand, all things are not only becoming connected to the Internet, but are also increasingly become equipped with sensors, actuators, and the processing required for closing the loop in terms of intelligence and autonomy. Such capabilities will allow new forms of communication to things, especially with humans.

The micro-level feature means to highlight and remind this peculiarity of intelligence: very small chunks of intelligence, spread all over the system, can enable the individual intelligence of any sort of devices, promoting coordination and interoperation among different entities.

MICRO-INTELLIGENCE FEATURE



Our approach aims to bring intelligence at the micro-level of the IoT system, so that micro-entities, i.e. Things, (agents, robots, humans' wearable devices,...), possibly with limited sensing and computation, "may operate intelligently in very large groups or swarms to affect the macroworld and reach the global and social intelligence" [MHH98].

Abstract, Reason, Plan, Solve, and Learn Historically, many approaches to Artificial Intelligence (AI) have been followed, each by different people with different methods and many definition of "what AI is" have been discussed [RN03]. What is relevant for micro-intelligence now, however, is not commit to a specific definition of intelligence, but to identify some basic features associated with intelligence so as to exploit them widely in pervasive systems. Therefore we concentrate on some general principles behind AI, in particular on its main features that can be summarised as *reason, plan, solve, abstract, and learn* capabilities —in other words [RN03], as the abilities to:

MICRO-INTELLIGENCE ABILITIES

(i) represent and manipulate intelligence

(ii) create new associations discovering new knowledge

(iii) make consistent reasoning, starting from premises deducing consequences.

Since Bob Kowalski's and Alain Colmerauer's opening of Logic Programming in the 1970s, LP has expanded in various directions and contributed to the development of many other areas of Computer Science. LP has helped to place logic firmly as an integral part of the foundations of Computing and Artificial Intelligence. Logic programming was the enabling technology for new AI and 'knowledge processing' applications and it provided a unifying foundation for developments in AI, in programming languages, in formal methods for software engineering, and in parallel computing.

In the mid 1980's, Carl Hewitt argued that Logic Programming (LP) was inadequate for modelling open systems [Hew90]. Hewitt's objections rest on classical logic's use of a static, globally consistent system which cannot represent dynamic activities, inconsistencies, and non-global concerns. At the time, his broadside was addressed by two papers. Kowalski [Kow85] agreed that model theoretic formulations of logic were lacking, and proposed the use of knowledge assimilation to capture change, along with additional elements to deal with belief systems. Kowalski's subsequent work on the event calculus and reactive and rational agents [KS96, KS99] can be viewed as developments of these ideas. Kahn and Miller suggested concurrent LP as the logical framework for open systems [KM88]. Another way of replying to Hewitt's objection is to look at the new era of pervasive open system. The survey [Dav02] shows that LP is a successful component of Internet programming languages, employed for tasks ranging from security semantics, composition of heterogeneous data, to coordination 'glue'. Many approaches have moved beyond first order logic (e.g. to concurrent constraints, linear logic, higher order), and LP is frequently combined with other paradigms (e.g. mutable state, objects). Furthermore, many of the programming concerns for the Internet are still less than fully understood: for example, there are unanswered problems related to mobility, security, and failure. Pervasive systems have no global time or state (knowledge base). Administrative domains mean there is no single naming scheme. Control/coordination is decentralised: to avoid bottlenecks in communication, because it is more scalable, and due to the impossibility of organising a single locus of control or coordination.

According to [Dav02], LP is a natural choice when a programming task requires symbolic manipulation, extended pattern matching, rule-based representation of algorithms, inference/deduction, a high degree of abstraction, and notation which reflects the mathematical basis of computation. So, it is not surprising that LP languages have found wide usage in the Internet domain and seems an ideal way of distributed intelligence in IoT things.

This is why we select logical models and mechanisms to investigate its potential as micro-intelligence sources in IoT pervasive scenarios.

Accordingly, in the LP context, micro-intelligence could build over typical features of computational logic, such as the representation of knowledge in terms of First-Order-Logic, FOL in the following, (*abstract*), the ability to make valid inferences and deduction, and more generally to demonstrate theorems given a FOL theory (*reason, plan, solve, learn*). Consistently, our definition of *micro-intelligence in the LP world* builds on these abilities by providing

LP MICRO-INTELLIGENCE ABILITIES

(i) the possibility of programming devices with logical theories,

(ii) logical inference engines, and

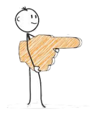
(iii) the possibility of exploiting logical demonstration and deduction.

Then, on the basis of the asserted knowledge it should be possible to *(i)* automatically derive new knowledge about the current context, and *(ii)* detect possible inconsistencies in the context information. With respect to *(i)*, reasoning aims to infer new context information based on the facts and on the information retrieved from sensors and other context sources—for instance, to derive the set of individual objects that are related to a given one by a particular property (e.g., the set of activities taking place in a specific location), or to calculate the most specific class an individual object belongs to (e.g., the fact that the activity performed by a given employee is a business meeting). With respect to *(ii)*, new possible scenarios –including automatic consistency checking– could be supported to capture possible inconsistencies in fragmented knowledge by exploiting model checking techniques that can be applied to check compliance rules.

Situatedness feature After Brooks' work [Bro91], another fundamental feature of AI is being *reactive* and *situated*. Our definition of micro-intelligence means to capture this aspect as the ability to cope with possible changes in perceived environment.

In case of LP models and technologies, the situatedness can thus be defined as:

LP
MICRO-INTELLIGENCE
FEATURE



the ability of logic theories, queries, and resolutions, to be context-aware w.r.t. (computational) environment, space, and time

The concept of situatedness has been used extensively since the mid-1980s in the cognitive science and AI literature, in terms such as ‘Situated Action’ [Suc87], ‘Situated Cognition’ (e.g., [Cla97]), ‘Situated AI’ (e.g. [HHC93]), ‘Situated Robotics’ (e.g., [HMB$^+$94]), ‘Situated Activity’ (e.g., [HJ96]), and ‘Situated Translation’ [Ris02]. Roughly speaking, the characterisation of an agent as *situated* is usually intended to mean that its behaviour and cognitive processes are first and foremost the outcome of a close coupling between agent and environment. Hence, situatedness is considered nowadays by many cognitive scientists and AI researchers a *conditio sine qua non* for any form of intelligence, natural or artificial.

A fundamental issue in the engineering of self-organising systems is the so-called “local-to-global” issue—that is, how to “link” the *local* mechanisms, through which the components of the system interact, to the *emergent, global* behaviour, exhibited by the system as a whole [BB06]. The exploitation of a situated LP approach, based on a common language of the knowledge, tries to give an answer to this issues, promoting at the same time the local inference and reasoning –concerning the local view of the system, and without any constraint of consinstency– and the global activities of more intelligent entities to gain the collective intelligence.

Overall, our approach means to provide distributed intelligence, direct or indirect interactions among relatively simple agents, flexibility, and robustness, in the perspective of the so-called *swarm intelligence* [Ken06]—the emergent collective intelligence of groups of simple agents, relying on the exploitation of fragmented knowledge with no attempt to put all the pieces together. There, huge numbers of small unit of computation (with inferential capabilities), situated within a spatially-distributed environment and promoting the local exploitation of high-level symbolic languages, collaborate to produce the social intelligence.

1.1.1 LPaaS as Micro-Intelligence

This thesis aims at exploring the effectiveness of an LP approach for modelling the micro-intelligence in the IoT world. In particular we propose theLPaaS model, where all the above LP features are made available in terms of service in the system.

In such a vision, the LPaaS abstraction represents a form of micro-intelligence, enabling situated reasoning, interaction and coordination in distributed systems, as the process by which an entity reasons about its local actions and the (anticipated) actions of others to try and ensure the community acts in a coherent manner. The LPaaS approach makes one step further, providing *intelligence as a service* – according to a SOA interpretation of the term service –, enabling people to benefit from ubiquitous information access. Emphasis is on on-demand applications, where the enabling infrastructure – servers, storage, networks, and client devices –moves towards cloud computing. LPaaS means to enable reasoning in distributed systems taking into account the explicit definition of the spatial-temporal structure of the environment in which situated entities act and interact, thus promoting an approach based on key features descending from the inner nature of pervasive systems like coordination, interaction and environment awareness.

1.1.2 LVLP as Micro-Intelligence

Specific local domains, however, may not be easily addressed by traditional logic description, in particular it could be difficult to model individual differences and the domain-specific knowledge of peculiar situations. Since *domain specific concepts* are necessary to model the ambient context, we complete our proposal with the *Labelled Variables in Logic Programming* (LVLP). The approach is consistent with the artificial intelligence literature, which has nearly universally determined that domain-specific knowledge is a major determinant of the success of expert systems [Cre93].

The LVLP purpose is to enable diverse computational models, each one tailored to the specific needs of situated components, to coherently and fruitfully coexist side by side, interacting within a logic-based framework —thus guaranteeing a shared and common understanding of some domain that can be communicated between people and application systems.

1.2 Structure of the Thesis

Accordingly, the contribution of this thesis may be conveniently organised in four main parts:

Part I (Chapters 2-3) addresses the intelligence issue in pervasive systems discussing the complementary research contributions necessary both to conceptually ground the LPaaS model, and to design & implement the LPaaS framework. First we review the literature historical perspective, analysing the evolution of the distributed LP approaches proposed over the time; then, we introduce and frame the LPaaS model and architecture.

Part II (Chapters 4-5) addresses the situatedness and domain-specific issues in logic programming, describing the state of the art and discussing how the LVLP model and technology could be seen as a possible answer.

Part III (Chapters 6-7-8) develops the LPaaS and LVLP technologies, rooted on the `tu`Prolog system, and discusses some experiments in Smart Environments, designed mostly to prove the feasibility and effectiveness of the above approach.

Part IV provides final remarks and a glimpse possible next steps.

More precisely, each Part starts with an overview Chapter and then goes in depth in its topics in the next Chapter(s).

Part I Chapter 2 first provides a review of the literature regarding the history and evolution of intelligence in distributed system, focussed on the models and technologies which mostly influenced the thesis: in particular, the intelligence issue is first addressed from the point of view of the distribution, then from the pervasive systems perspective. Finally, the way classical distributed LP notions need to be rethought in the IoT perspective is introduced.

Then, Chapter 3 introduces the notion of *Logic Programming as a Service* (LPaaS) as a means to address the needs of pervasive intelligent systems through logic engines exploited as a distributed service, it thoroughly describes the LPaaS model for spreading intelligence in pervasive systems. First the vision and the abstract architectural model are described by re-interpreting classical LP notions in the IoT context, then the nature of LP interpreted as a service is discussed by describing the basic LPaaS interfaces.

Part II Chapter 4 is devoted to address the Situatedness & Domain-specific issue in Logic Programming. In particular, the state of the art is discussed, focussing on the models and technologies which mostly influenced the thesis.

Chapter 5 presents the *Labelled Variables in Logic Programming* (LVLP) extension to enable LP to deal with the diversity of pervasive systems, where many heterogeneous, domain-specific computational models could benefit from the power of symbolic computation. The model for labelled variables in logic programming is defined. The fixed-point and the operational semantics are also introduced, discussing the correctness, completeness, and their equivalence. Finally, an implementation on the top of the tuProlog system is presented and discussed.

Part III Chapter 6 presents the tuProlog environment, consisting of a prototype of both LVLP and LPaaS and a complete eco-system currently under development.

Chapter 7 explores the effectiveness of merging the two above proposed approaches, LPaaS and LVLP, to promote situated distributed intelligence in pervasive systems and in particular in Smart Environments. After reviewing the literature, the Butlers for Smart Spaces approach, a technology-neutral reference framework focused on users' situatedness and interaction aspects, is described.

Chapter 8 reports on some experiments of the integration of the two presented technologies in Smart Environments deployed on the Home Manager platform, designed to prove the feasibility and test the effectiveness of the above approach. The technology is discussed, describing both a prototype middleware and a complete eco-system currently under development. Some running example are presented and discussed.

Part IV Part IV concludes the thesis with final remarks and outlining a research roadmap for further work, about each of the research lines brought by the thesis.

Table 1.1 lists the main acronyms used in this book.

Acronym	Meaning
LP	Logic Programming
FOL	First-Order-Logic
DSL	Domain-specific languages
IoT	Internet of Things
IoIT	Internet of Intelligent Things
SOA	Service-Oriented-Architecture
MAS	Multi Agent System
LPaaS	Logic Programming as a Service
LVLP	Labelled Variables in Logic Programming
SE	Smart Environment
STS	Socio-Technical System
KIE	Knowledge-Intensive Environment
MCS	Multi-Context System

Table 1.1: *Table of Acronyms*

1.3 List of Publications

Here follows a comprehensive list of the publications which directly contributed to the body of work presented in this thesis, authored, or co-authored, by the same author of this thesis:

Chapter 3

- Roberta Calegari, Enrico Denti, Stefano Mariani, and Andrea Omicini. Logic Programming as a Service (LPaaS): Intelligence for the IoT. In Giancarlo Fortino, MengChu Zhou, Zofia Lukszo, Athanasios V. Vasilakos, Francesco Basile, Carlos Palau, Antonio Liotta, Maria Pia Fanti, Antonio Guerrieri, and Andrea Vinci, editors, *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC 2017)*. IEEE, May 2017
- Roberta Calegari, Enrico Denti, Stefano Mariani, and Andrea Omicini. Towards logic programming as a service: Experiments in tuProlog. In Corrado

Santoro, Fabrizio Messina, and Massimiliano De Benedetti, editors, *WOA 2016 – 17th Workshop "From Objects to Agents"*, volume 1664 of *CEUR Workshop Proceedings*, pages 91–99. Sun SITE Central Europe, RWTH Aachen University, 29–30 July 2016. Proceedings of the 17th Workshop "From Objects to Agents" co-located with 18th European Agent Systems Summer School (EASSS 2016)

- Enrico Denti, Andrea Omicini, and Roberta Calegari. tuProlog: Making Prolog ubiquitous. *ALP Newsletter*, October 2013

Chapter 5

- Roberta Calegari, Enrico Denti, Agostino Dovier, and Andrea Omicini. Extending logic programming with labelled variables: Model and semantics. *Fundamenta Informaticae*, 2017. Special Issue CILC 2016
- Roberta Calegari, Enrico Denti, Agostino Dovier, and Andrea Omicini. Labelled variables in logic programming: Foundations. In Camillo Fiorentini and Alberto Momigliano, editors, *CILC 2016 – Italian Conference on Computational Logic*, volume 1645 of *CEUR Workshop Proceedings*, pages 5–20, Milano, Italy, 20-22 June 2016. CEUR-WS. Proceedings of the 31st Italian Conference on Computational Logic
- Roberta Calegari, Enrico Denti, and Andrea Omicini. Labelled variables in logic programming: A first prototype in tuProlog. In Elena Bellodi and Alessio Bonfietti, editors, *Proceedings of the Doctoral Consortium of the 14th Symposium of the Italian Association for Artificial Intelligence (AI*IA 2015 DC)*, volume 1485 of *CEUR Workshop Proceedings*, pages 25–30, Ferrara, Italy, 23–24 September 2015. AI*IA, CEUR-WS

Chapter 7-8

- Roberta Calegari and Enrico Denti. *Context Reasoning and Prediction in Smart Environments: The Home Manager Case*, pages 451–460. Springer International Publishing, Cham, 2018. Proceedings of IIMSS 2017, Vilamoura, Portugal, 21-23 June 2017
- Roberta Calegari and Enrico Denti. Building Smart Spaces on the Home Manager platform. *ALP Newsletter*, December 2016
- Roberta Calegari and Enrico Denti. The Butlers framework for socio-technical smart spaces. In Franco Bagnoli, Anna Satsiou, Ioannis Stavrakakis, Paolo Nesi, Giovanna Pacini, Yanina Welp, Thanassis Tiropanis, and Dominic DiFranzo, editors, *Internet Science. 3rd International Conference (INSCI 2016)*, volume 9934 of *LNCS*, pages 306–317. Springer, 2016

- Enrico Denti and Roberta Calegari. Butler-ising HomeManager: A pervasive multi-agent system for home intelligence. In Stephane Loiseau, Joaquim Filipe, Beatrice Duval, and Jaap Van Den Herik, editors, *7th Int. Conf. on Agents and Artificial Intelligence (ICAART 2015)*, pages 249–256, Lisbon, Portugal, 10–12 January 2015. SCITEPRESS
- Enrico Denti, Roberta Calegari, and Marco Prandini. Extending a smart home multi-agent system with role-based access control. In *5th Int. Conf. on Internet Tech & Society*, pages 23–30, Taipei, Taiwan, 10–12 December 2014. IADIS Press. Best Paper Award

Part I

Intelligence Issue in Pervasive Distributed Systems

In the first part of this thesis the engineering challenges posed by three different but related kinds of complex systems are discussed from a coordination models & technologies perspective, then a novel approach to deal with each of them is presented. In particular, the focus is on the history and evolution of *intelligence & distribution* and *intelligence & situated pervasive* systems (Chapter 2).

Each contribution is a necessary ingredient for building the LPaaS model described in Chapter 3 of this thesis, a novel approach intended as the natural evolution of distributed LP in pervasive systems, explicitly designed to exploit context-awareness so as to promote the distribution of situated intelligence within smart environments. Namely, the approach has to deal with distribution in that it aims to deliver intelligence *as a service*, granting ubiquitous access to knowledge and on-demand reasoning via LP services, spread over the network and configured to respond to specific local needs. Similarly, *time and space situatedness* enable clients to submit *situated* queries where the notions of time and locus explicitly intervene in the computation.

Chapter 2

History and Evolution

In this chapter the issue of intelligence in pervasive distributed system is addressed, providing an overview of the main research approaches during the last decades. In particular, since Logic Programming (LP) languages and technologies represent the most natural candidates for injecting intelligence within computational systems, the evolution of distributed logic programming is retraced. Then the classical distributed LP notions are re-framed in the IoT perspective, from both the engineering standpoint of *interaction*, and from the standpoint of *architecture*.

As discussed in the Introduction, computation is moving towards pervasive, ubiquitous environments where devices, software agents, and services are expected to seamlessly integrate and cooperate in support of human users, anticipating their needs and more generally acting on their behalf, delivering services in an "anywhere, anytime" fashion [FJK$^+$01, ZOA$^+$15].

The above scenarios are naturally fit for a distributed approach: tasks are often distributed in space, time, or functionality, and can benefit from the chance of solving subproblems modularly and concurrently. At the same time, these scenarios inherently call for intelligence – namely, *distributed situated intelligence* [Par08] – to exploit domain knowledge, understand local context, and share information in support of intelligent applications and services [CFJ03, Sma17].

Accordingly, the issue of distributed situated intelligence is addressed, firstly taking into account the evolution and the history of distributed logic programming in Section 2.2, then, reporting the state of the art about dealing with intelligence in pervasive system (Section 2.1). Finally, as a natural evolution of these works, Section 2.3 introduces the key features behind LPaaS, by discussing how the service perspective and the new situated dimension of computation mandate for a re-interpretation of some basic LP concepts.

2.1 Intelligence in Situated Pervasive Systems

Distributed Artificial Intelligence (DAI) is concerned with the study and construction of semi-autonomous automated systems that interact with each other and their environments. It goes beyond the study of individual intelligent agents solving individual problems, towards problem solving that has *social* components. With the advent of large computer and telecommunications networks, the problem of integrating and coordinating many human and automated problem solvers working on multiple simultaneous problems has become a pressing concern [Ros94]. Just as "conventional" AI research has sometimes used individual human psychology or cognition as a model or driving metaphor, DAI considers concepts such as group interaction, social organisation, and society as metaphors and problem generators [Ros94]. Highly-organised DAI systems are now a research reality, and are rapidly becoming practical partners in critical human problem-solving environments.

As defined in [GH90], researchers would most like to have a theory which relates features of domain problems and knowledge organisation to choices on modelling, implementation, and performance questions, but this theory is incomplete. Some criteria for DAI applications problems, that help identify domains in which coordination among intelligent agents is a basic issue, include:

- clear (possibly hierarchical) structure of time, knowledge, communication, goals, planning, or action
- natural (not forced) distribution of actions, perceptions, authority, and/or control
- interdependence because local decisions may have global impacts, and possible harmful interactions among agents
- possible limits on communication time, bandwidth, etc., so that a global view-point, controller, or solution is not possible.

Increasing contextual awareness could help to face this issue; in fact in [DLC87], agents can approximate the accuracy of centralised reasoning if they are provided with more knowledge about other problem solvers in order to reason about potential conflicts in knowledge, goals, plans, and activities. Promising techniques include incrementally expanding local views based on causal plan relations, as in multistage negotiations [CKLM91].

Research into context and context-awareness (generally) focuses on relatively simple context-aware applications using principally spatio-temporal and identity information. Current research aims to develop generic context models with "representation and query languages and context reasoning algorithms" [SLP04] to facilitate context sharing and interoperability of applications. The context modelling approaches are classified in [SLP04] under six major headings:

- Key-Value Models (KVM)
- Markup Scheme Models (MSM)
- Graphical Model (GM)
- Object-Oriented Models (OOM)
- Logic-Based Models (LBM)
- Ontology-Based Models (OBM)

KVM The KVM approach is used in [SAW94] to propose: "an extended form of mobile computing in which users employ many different mobile, stationary and embedded computers over the course of a day" and PARCTab [WSA+95]. The model operates on the basis that computation does not occur at a single location or in a single context, as in desktop computing, but covers a number of situations, locations, and environments.

MSM A markup language combines text with additional descriptive information. The best-known markup languages in modern use are the Hypertext Markup Language (HTML) and the Extensible Markup Language (XML). Markup languages form the basic components of MSM which are characterised by a hierarchical data structure using a combination of tags with attributes and content, the attributes are context properties [MH06].

OOM A common feature of OOM approaches is the aim of achieving the (principal) benefits of object oriented approaches which are (1) encapsulation and (2) reusability [SP99] to address issues arising from the dynamics of context in ubiquitous environments. Context processing [WSA+95] is encapsulated into an object level with access to contextual information through specified interfaces.

LBM Logic addresses scenarios in which an expression or fact(s) may be derived from a set of expressions or facts. Formal systems use inference rules to support such reasoning [MH07]. In logic-based context models, a context is (generally) defined using facts (context properties) with expressions and rules to describe and define relationships and constraints. Contextual information is generally added, updated or deleted in terms of facts or is inferred using rules that describe and define relationships and constraints in logic-based systems.

OBM Ontologies represent a valid approach with which to specify concepts and relationships [UG96, Gru93, HPSvH03] being particularly suitable to represent contextual information in machine readable form in a data structure such as RDF/S with OWL [MH07, ADB+99]. The preceding context modelling approaches can all be viewed as precursors to OBM, ontologies incorporating and utilising many of the concepts that characterise the other modelling approaches.

Relation with multi-context systems The Multi-Context System (MCS) framework [BA10, BRS07, GG01, SH12], which evolved from MultiLanguage systems [Giu92], is an expressive, uniform, high-level framework for the integration of heterogeneous knowledge sources in a modular and very general way. It is a powerful knowledge representation formalism for many application scenarios where heterogeneity and inter-contextual information exchange are essential. More concretely, in MCS decentralised and heterogeneous system parts interact via (possibly non-monotonic) bridge rules for information exchange.

Being mainly designed for static scenarios, however, MCS are not well suited for dynamic environments characterised by streams of constantly-arriving data [BEG+18], which should make it possible to reason continuously over such heterogeneous knowledge. More generally speaking, our approach is different in that we do not intend to interlink and build consistency between all the distributed logic theory. Instead, we move from the assumption that each situated node should have its partial knowledge, possibly even inconsistent with the whole picture, but sufficient for partial reasoning on that specific situation/location.

2.2 Intelligence in Distributed Logic Programming Systems

Research on distributed intelligence has gained increasing popularity over the years [Par08]. Starting from the seminal work of [CG81], concurrency, parallelism, and several approaches for distributing intelligence have been explored—from LP languages specially designed for distribution, to pure logic-based models, rule-based systems, probabilistic graphical models, and ontologies. In the following we organise and describe some of the most relevant contributions to the field, focussing on those that mostly relate to our approach in the effort of motivating the need for further advancement.

Implicit Parallelism. The first efforts to advance beyond sequential LP start from the programming schemes for the interpretation of logic programs—in particular, towards implicit parallel evaluation, leading to explore AND-parallelism, OR-parallelism, Search parallelism, and Stream-AND-parallelism.

[Rei78] introduces a scheme that allows negative literals in queries; some years later, the Naish scheme [Nai88] introduces co-routing among procedure calls. Meanwhile, [WML84] focuses on AND-parallel evaluation: their asynchronous version corresponds to the execution models of parallel LP languages. These schemes perform and adapt well to different forms of parallelism: altought, they were not meant to face distributed programming.

Also, it is worth noting that implicit parallelism lacks two important control mechanisms: synchronisation of logic processes, and control over non-determinism of schedulers.

Explicit Parallelism. Later approaches focus on “extraction” of parallelism via explicit language constructs.

A first research line moves from concurrent logic languages, rooted in Relational Language [CG81], generally acknowledged as the first concurrent LP language. In Concurrent Prolog [Sha87], Guarded Horn Clauses [Ued86], and Parlog [Cla87], goal evaluation is carried out by a network of fine-grained logic processes (i.e., atomic goals) that are executed in parallel: processes communicate via shared streams, i.e., bi-directional channels on which data items flow.

An alternative research line follows the idea of extending Prolog with special features for distributed execution, like message passing. This approach preserves the operational semantics of sequential Prolog, augmenting the language with ad-hoc communication primitives. One of the major references in this field is Delta Prolog [BG89], where Prolog is extended with constructs for sequential and parallel composition of goals, inter-process communication and synchronisation, and external non-determinism. Delta Prolog programs using concurrency mechanisms [CFP89] do not lend themselves to the usual declarative interpretation as Horn clauses, but are grounded on the theory of Distributed Logic [Mon84]. This approach extends Horn clause logic with the notion of time-dependent events, on which process communication and synchronisation are based, making distributed logic a special kind of temporal logic.

Besides enabling inter-process communication for logic programs, orthogonal aspects such as their deployment are not considered, neither are the issues brought along by distribution taken into account—such as validity in time of logic theories and their global consistency.

Agents, Communication, and Coordination for Distributed LP. Further steps towards distributed LP came with Shared Prolog [BC91], based on parallel *agents* that are Prolog programs extended with a guard mechanism. The programmer controls the granularity of parallelism, coordinating agents' communication and synchronisation via a centralised data structure, the *blackboard*, inspired to the omonymous model [Nii86] as well as to the LINDA coordination model [Gel85]. The main idea is to exploit the blackboard within the logic framework to coordinate logic processes. However, the inference engine is not situated in time and space, i.e., the query result is independent from the position of the entities, flow of time, and context/situation changes.

LP in Pervasive, Context-aware Systems. More recently, LP has been explored as a promising solution to bring intelligence into pervasive context-aware systems.

[RC03] shows that first-order logic is a very effective and powerful way of dealing with context, promoting an approach to develop a flexible and expressive model supporting context-awareness, thus enabling deduction of higher-level situations from perceptions about basic contexts—through rule-based approaches. A key advantage of a formal model for context is that the expressiveness of the model can be clearly specified and automatically verified. [Lok04] emphasises that LP is generally useful for context reasoning, as well as for supporting rule-based (meta)programming in context-aware applications, enabling, i.e., hierarchical description of complex situations in terms of other situations. The approach encourages a high-level of abstraction for representing and reasoning about situations, and supports building context-aware systems incrementally through modularity and separation of concerns. The focus on context-awareness of both contributions motivates our focus on re-interpreting distributed LP targeting especially context-aware systems, as pervasive ones usually are—being the IoT a prominent example.

Other works take different approaches from pure logic-based models to rule-based systems and probabilistic graphical models, up to ontologies.

Rule-based systems [SDA99, Dey01, ECB06, WMSC11] have been in use for decades for both model representation and reasoning in context-aware applications. More recently, [NB14] have proposed a rule-based, learning middleware for storage and reasoning in a distributed scenario. The idea is to delegate context acquisition to the middleware, that is, a rule-based context reasoning platform tailored to the needs of intelligent distributed mobile computing devices. The need for a dedicated middleware layer is apparent in the aforementioned work, further strengthening the idea that distributed LP is not confined to context manipulation, but deserves general attention.

In [RAMC04], fuzzy and probabilistic logic is exploited to handle uncertainty of the environment and to deal with the imperfections of data. Probabilistic graphical models [BBH$^+$10] can be exploited to support the modelling of, and the reasoning about, uncertain information in pervasive systems, even if exact inference in complex probabilistic models can be a NP-hard task. Description logic, usually used in combination with ontologies, is another LP extension that has proven effective for modelling concepts, roles, individuals, and their relationships, as well as to provide simple reasoning capabilities [HWD12]. However, only simple classification tasks can be solved, and no mechanisms are provided to infer more complex information from existing data [NB14]. Also, design and implementation are typically more difficult and time-consuming than with other approaches. Uncertainty of information is the natural enemy of global consistency, thus our approach considers to abandon the idea of globally (in terms of both time and space) consistent logic theories (or, knowledge bases—KB) in favour of locally consistent ones.

2.3 Re-thinking classical distributed LP notions in the IoT perspective

The evolution of LP in parallel, concurrent, and distributed scenarios is the main motivation for re-interpreting the notion of *distribution* of LP in today's context.

Logic programming boasts a long-respected reputation in supporting intelligence: originally conceived for single solvers and later extended towards concurrency and parallelism, LP has the potential to fully support pervasive computing scenarios once it is suitably re-interpreted. The re-interpretation of LP should develop along three main lines: *(i) architecture*—that is, the need to go beyond the (originally monolithic) structure of LP systems, which is unsuitable for distributed contexts such as IoT mobility/cloud ecosystems, typically grounded on the service-oriented computing paradigm [Dug12, Erl05]; *(ii) situatedness*—that is, the ability of logic theories, queries, and resolutions, to be context-aware w.r.t. (computational) environment, space, and time; *(iii) interaction*—that is, the opportunity to re-think the interaction patterns used by clients to query logic engines, which should lean towards on-demand computation.

At the same time, the declarativeness and the explicit knowledge representation of LP enable knowledge sharing an adequate level of abstraction while supporting modularity and separation of concerns [OP11], which are especially valuable in open and dynamic distributed systems (*serendipitous interoperability*, [Nie13]). As a further element, its sound and complete semantics naturally enables intelligent agents to reason and infer new information. Finally, specific LP extensions or logic-based computational models – such as meta-reasoning about situations [Lok04] or labelled variables systems [CDDO16] – could be incorporated so as to enable complex behaviours tailored to the needs of situated components.

Our approach promotes key properties of pervasive systems as observability, malleability, understandability, formalisability, and norm compliance. The service behaviour follows the specification for which it has been forged: given that specification and the history, the dynamic behaviour of the system can be observed, understood and somehow predicted.

2.4 Remarks & Outlook

The material presented in this chapter is at the very core of the *Logic Programming as a Service* (LPaaS) model described in Chapter 3. In particular:

- our approach is intended as the natural evolution of distributed LP in pervasive systems, explicitly designed to exploit context-awareness so as to promote the distribution of situated intelligence within smart environments
- client/service interaction is no longer bound to the traditional console-based query/response loop, but is redesigned to provide the dynamism, flexibility, and expressiveness required by the targeted application scenarios—e.g., IoT
- time and space situatedness is taken into account from the design, promoting novel forms of client/service interaction, enabling clients to submit "situated" queries where the notions of time and locus explicitly intervene in the computation.

Next chapter presents the abstract architectural model, conceived and designed to better support the approach here described to engineering intelligence, within pervasive computing scenarios.

Chapter 3

Logic Programming as a Service

In this Chapter moving from the comparison of the approaches and techniques discussed in chapter 2, we define a new paradigm, namely Logic Programming as a Service (LPaaS), as the natural evolution of distributed LP in nowadays pervasive systems to support context-aware distributed intelligence in smart environments (e.g. intelligent meeting rooms, smart homes and smart vehicles). The new proposed approach could be effective for designing and developing distributed context aware systems, keeping into account the advantages of previous research in this area while giving a re-interpretation of classical LP notion in a novel pervasive contex-aware perspective. Thus, LPaaS starts from requirements resulting from the actualisation of the discussed approaches.

The LPaaS abstraction enables *situated reasoning*, *interaction* and *coordination* in distributed systems, as the process by which an entity reasons about its local actions and the (anticipated) actions of others to try and ensure the community acts in a coherent manner. It promotes the interactions, coordination and cooperation of numerous, casually accessible, and often invisible computing devices of pervasive systems, particularly taking into account the localised and dynamic nature of interaction.

The LPaaS is a paradigm that exploits the strengths of service oriented architectures but it incorporates an idea of a mobile device in a distributed system as an autonomous context-aware entity, equipped with intelligent and context-aware inference service. The architecture offers context-based reasoning to many applications at the same time, enforcing the interoperability between different entities in order to reach a social desired behaviour.

Accordingly, Section 3.1 introduces the vision behind LPaaS, by discussing how the service perspective and the new situated dimension of computation suggest for a re-interpretation of some basic LP concepts. Section 3.2 shows how such a re-interpretation affects LP at the architectural level, by discussing the logic-based service-oriented architecture supporting LPaaS more practically. Section 3.3 defines the LPaaS service interface, and elaborates on the interaction patterns. In Section 3.4 some prototype implementation developed on the top of the `tu`Prolog system are discussed and tested in some simple case studies.

3.1 Vision

Since SOA are the *de facto* standard for distributed application development in both the academia and the industry [GS12], our starting point is how LP can be re-interpreted in the *service* perspective (Subsection 3.1.1). The service perspective further emphasises the role of *situatedness*, already brought along by distribution in itself: thus, Subsection 3.1.2 discusses how being situated in space, time, and context affects LP computation. The two novel perspectives are brought together in Subsection 3.1.3, which develops the idea of LP as a *situated service*.

3.1.1 The Service Perspective

The service-oriented perspective deeply affects the way in which LP engines are conceived, designed, and used—in particular, as far as the very nature of LP *encapsulation*, the way in which clients interact (requiring *statelessness*), and the assumptions about the surrounding context (*locality*) are concerned.

Encapsulation. A service hides both data representation and the computational mechanisms behind a public interface exposed to its clients. In the context of LP engines, this means that both the logic theory (the data) and the resolution process (the computational mechanism) are *inaccessible* and in general *not observable* from outside the boundary of the service interface. As a consequence, theory manipulation mechanisms, such as `assert`/`retract`, should be no longer directly applicable from the client perspective: since the logic theory is the data encapsulated by the service, dedicated mechanisms are required for its handling. For instance, in an IoT scenario, this could happen via a separate "sensor API" through which sensor devices update the KB of the LP service according to their perception of the environment.

Accordingly, the logic theory of a LPaaS service can be either *static* or *dynamic* (which are mutually exclusive configurations), affecting the way in which the LP service can be accessed obviously depends on that —namely, time is an issue for dynamic KB, not for static ones.

Statelessness. Encapsulation makes it irrelevant *how* the encapsulated behaviour is implemented: what actually matters are the inputs and outputs triggering and resulting from that behaviour. Furthermore, in the SOA perspective, services are usually redundantly distributed over a network of hosts for enhancing the service availability and reliability: thus, it doesn't really matter *who* actually carries out the encapsulated behaviour. In the context of LP, this means that interactions with clients should be allowed to be *stateless*—that is, include all the information required by the resolution process, since a different component may serve a different request. Notably, stateless interaction is preferred for RESTful web services, too [FT02].

Locality. The distributed nature of the system drastically changes the perspective over consistency: maintaining *globally-consistent* information is typically unfeasible in such systems. Furthermore, when pervasive systems come to play, even *globally-available* information is usually not a realistic assumption: for instance, in IoT scenarios, heterogeneous data streams are continuously made available by sensor devices scattered in specific portions of the physical environment. As a consequence, encapsulation is inevitably bound to a specific, (*local*) portion of the system—with a notion of locality extending up to when/where availability and/or consistency are necessarily lost.

In the context of LP, this means first of all turning to a *multi-theory* logical framework, exploiting the typical approach to *modularity* adopted in traditional LP in order to allow for parallel and concurrent computation [BLM94]. Then, locality also implies that each logic theory describes just what is *locally* true —which basically means leaving aside in principle the global acceptation of the *closed world* assumption [Rei78] in favour of a more realistic *locally-closed world* assumption. Accordingly, every LP service is to be queried about *what is locally known to be true*, with no need to have a global knowledge of any sort—and with no need to distribute the resolution process in any way.

3.1.2 The Situatedness Perspective

The distribution of LP service instances directly calls for *situatedness*, intended as the property of the LP service to be immersed in the surrounding computational/physical environment, whose changes may affect its computations [MO15]. As an example, new sensor data may change the replies of an LP service to queries. Situatedness adds three new dimensions to LP computations: *space*, *time*, and *context*.

Space. To be situated in space means that the *spatial context* where the LP service is executing may affect its properties, computations, and the way it interacts with clients.

Distribution *per se* constitutes a premise to spatial situatedness: each LP instance runs on a different device, thus on a different network host, therefore accessing the different computational and network resources that are *locally* available. Moreover, since LP services encapsulate the logic theory for their resolution process, the locally-gathered knowledge affects the result, once it is represented in terms of logic axions.

Also, more articulated forms of spatial situatedness may be envisioned: for instance, *mobile* clients may request LP services from different locations at each request, possibly even *while* moving, which means that the LP service must be able to coherently identify and track clients so as to reply to the correct network address. Finally, it is possible in principle to conceive logic theories – or even individual axioms therein – with spatially-bound validity, that is, holding true in specific points or regions in space—analogously to *spatial tuples* in [RVO+17].

Time. Complementarily, being situated in time means that the *temporal context* when the LP service is executed may affect its properties, computations, and interactions with clients. Yet again, distribution alone already brings about temporal issues: moving information in a network takes time, thus aspects such as expiration of requests, obsolescence of logic theories, and timeliness of replies should be taken into account when designing the LP service.

Furthermore, since reconstructing a *global* notion of time in pervasive systems is either unfeasible or non-trivial, it is more likely that each LP service operates following its own local time, thus computing deadlines, leasing times, and similar according to its *local perception* of time. Also, in the same was as for spatial situatedness, temporal situatedness may also imply that logic theories or individual axioms may have their validity bound in time—e.g., holding true up to a certain instant in time, holding no longer from then on.

Context. Besides the space/time, situatedness also regards the generic *environment* within which LP services execute—that is, the computational and physical context which may affect their working cycle: for instance, it may depend on the available CPUs and RAM, whether an accelerometer is available on the current hosting device, etc.

A basic level of *contextual situatedness* is already embedded in the very nature of the LP service: in fact, locality of the resolution process implies that the logic theory for goal resolution belongs to the context of the LP service, affecting its behaviour. However, especially in the IoT scenarios envisioned for LPaaS, the computational and physical contexts may both impact the LP service: e.g., sensor devices may continuously update the service KB with their latest perceptions, while actuators may promptly provide feedback on success/failure of physical actions.

3.1.3 Towards LP as a Situated Service

The above perspectives promote a radical re-interpretation of a few facets of LP, moving LP itself towards the notion of LPaaS envisioned in this work—that is, in terms of a *situated service*. Such a notion articulates along four major aspects:

- the conservation (with re-contextualisation) of the SLD resolution process;
- stateless interactions;
- time-sensitive computations;
- space-sensitive computations.

The re-contextualisation of the SLD resolution process. The SLD resolution process remains a staple in LPaaS: yet, it is re-contextualised in the situated nature of the specific LP service. This means that, given the precise *spatial*, *temporal*, and *general* contexts within which the service is operating *when the resolution process starts*, the process follows the usual rules of SLD resolution: situatedness is accounted for through the service abstraction with respect to such three contexts.

With respect to the *spatial context*, the resolution process obviously takes place in the hosting device *where* the LP service is running, thus taking into account the specific properties of the computational and physical environment therein available – CPU, RAM, network facilities, GPS positioning, etc. – there included the specific logic theory the LP service relies on. As mentioned in Subsection 3.1.2, more complex forms of spatial situatedness, e.g. involving mobility of clients (and LP services, in principle), or virtual/physical regions of validity for logic axioms, could be envisioned.

The *temporal context* refers to the resolution process taking place on a *frozen snapshot* of the LP service state – there including its KB –, which stays unaffected to external stimuli (possibly affecting the resolution process) until the process itself terminates. This way, despite the dynamic nature of the KB – encapsulated by the service abstraction – which could change e.g. due to sensors' perceptions, the resolution process is guaranteed to operate on a consistent stable state of the logic theory.

Finally, the resolution process depends on the *general context* of the specific device hosting the LP service instance—thus considering the state of KB therein available, as assembled by e.g., the set of sensors devices therein available, the service agents gathering new local information, and so on.

Stateless interactions. A first change brought by LPaaS concerns the interaction with the clients of the LP service.

In classical LP, interactions are necessarily *stateful*: the user first sets the logic theory, then defines the goal, and then asks for one or more solutions, iteratively. This implies that the LP engine is expected to store the logic theory to exploit as its KB, to memorise the goal under demonstration, and to track how many solutions have been already provided to the user: all these information become part of the state of the LP engine.

Instead, in LPaaS interactions are first of all (even though not exclusively) *stateless*: coherently with SOA, the LP service instance that actually serves each request may be different at each time, e.g. due to redundancy of distributed software components aimed at improving availability and reliability of the LP service. In such a perspective, each client query (interaction) should be possibly self-contained, so that it does not matter which specific service instance responds—because there is no need for it to track the state of the interaction session.

Time-local computation. Another change stemming from the situated nature of LPaaS regards the relationship between the resolution process and the time flow.

In pure LP, the logic theory is simply assumed to be always valid, and time-related aspects do not affect the resolution process; for instance, assertion / retraction mechanisms are most typically regarded as extra-logic. As discussed above, in LPaaS the consistency of the resolution process is guaranteed by the fact that the possibly ever-changing KB encapsulated by the service is *frozen* in time when the resolution process itself begins: nevertheless, time situatedness requires by definition that time affects the LP service computation in some way.

Accordingly, in LPaaS each axiom in the KB is decorated with a *time interval*, indicating the time validity of each clause. Every time a new resolution process starts in order to serve a LPaaS request, the logic theory used is the one containing all and only the axioms holding true at the *timestamp* associated to the resolution process itself. In the simplest case, such a timestamp is implicitly assigned by the LP server as the current local time when the request for goal demonstration is first served. However, it could also be explicitly assigned by clients along with the request—e.g., specifying a specific time when asking for a goal demonstration.

Space-local computation. Analogously, classical LP has no notion of space situatedness: be it virtual or physical space, the LP engine is a monolithic component providing its "services" only locally, to its co-located "clients" executing on the same machine.

The LPaaS interpretation stems again from the very nature of service in modern SOA-based applications—a computational unit providing its functionalities through a network-reachable endpoint. Therefore, in LPaaS the resolution process is naturally and inherently affected by the specific *computational locus* where a given LP service instance is executing at a given moment—there including the locally-available resources.

3.2 Model and Architecture

Following the lines traced in Subsection 3.1.3, we now elaborate more practically on how encapsulation, statelessness, and locality – that is, the *service perspective* (Subsection 3.1.1) – are exploited in LPaaS according to the three dimensions of *situatedness* described in Subsection 3.1.2—that is, time, space, and context. Then, we briefly describe *microservices* [Fam15] as a key enabler architecture for LPaaS.

3.2.1 The Service Architecture

Encapsulation As it straightforwardly stems from SOA principles, encapsulation is exploited in LPaaS so as to define a standard API that shields LPaaS clients from the inner details of the service while providing suitable means of interaction.

Accordingly, each LP server node exposes its LP service to clients via two interfaces, depicted in Figure 3.1:

- **Client Interface:** exposes methods for *observation* and *usage*. *Client* refers to any kind of users, either individuals (humans, software agents) or groups entitled to exploit the LPaaS services.

- **Configurator Interface:** enables service *configuration* and requires proper access credentials. *Configurator* refers to service managers—privileged agents with the right of enforcing control and policies for that local portion of the system.

Applications can access the service as either *Clients* or *Configurators*, via the corresponding interfaces. The service is initialised at deployment-time on the server machine: once started, it can be dynamically re-configured at run-time by any configurator.

Locality. Situatedness is exploited as a means to consistently handle *locality* w.r.t. context, time, and space.

In fact, dealing with situated logic theories means first of all giving up with the idea of global consistency in a closed world: in LPaaS multiple KB are spread throughout a network infrastructure, likely geographically distributed, executing within different computational contexts, and possibly either fed by sensors or manipulated by service agents perceiving the physical context. Nevertheless, by allowing distributed access and reasoning over its own locally-situated knowledge base, each LPaaS node actively contributes to the overall availability of the global knowledge.

Accordingly, pervasive application scenarios where logic theories represent local knowledge inherently call for *dynamic KB*, *autonomously* evolving during the service lifetime. Here,"autonomously" means that in the LPaaS perspective the logic KB may evolve over time with no need for a client to invoke `assert/retract`, or equivalent methods – which, in fact, are not included in the LPaaS standard API detailed in Subsection 3.3.1 – but, i.e., due to sensor devices' perceptions transparently feeding the LP service KB. As such, each situated KB of a LPaaS service can be seen as representing what is known to be true and relevant in a given location in space at a given time, thus possibly changing over time – e.g., due to data streams coming from sensor devices –, and potentially different from any other KB located elsewhere—as depicted in Figure 3.2. Accordingly:

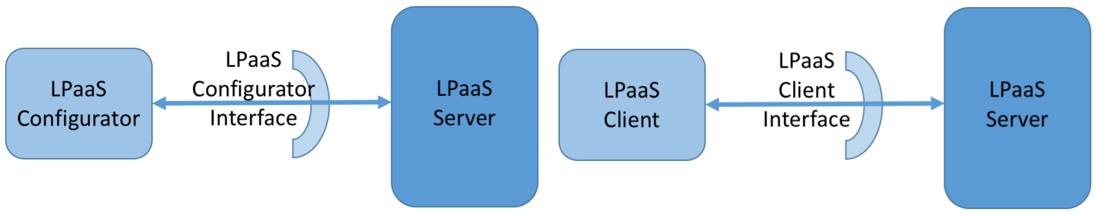


Figure 3.1: *LPaaS Configurator Service Architecture (left) and Client Service Architecture (right)*

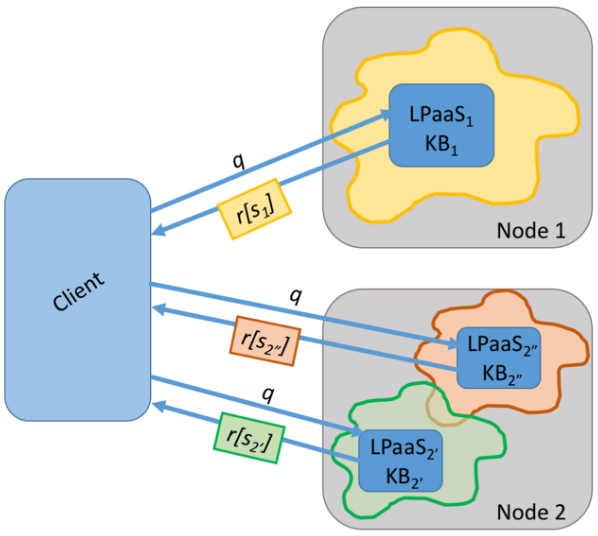


Figure 3.2: *Situatedness of LPaaS: the same query (q) by the same client may be resolved differently ($r[s_1], r[s_{2'}], r[s_{2''}]$) by distinct LPaaS services ($LPaaS_1, LPaaS_{2'}, LPaaS_{2''}$) based on their local computational, physical, and spatio-temporal context ($S_1, S_{2'}, S_{2''}$)*

- each LPaaS clause has a lifetime, expressed as a time interval of validity—as in the case of the “current” temperature in a room
- as a result, at any point t in time a LPaaS service has precisely one logic theory made of all and only the clauses that hold true at time t
- each LPaaS resolution process is either implicitly (by the LPaaS server) or explicitly (by the LPaaS client) labelled with a *timestamp*, used to determine the KB to be used for the resolution itself—which then works then as the standard LP resolution

Statelessness. *Uncoupling* is one of the main requirements for interaction in distributed systems: that is why LPaaS provide stateless client-server interaction as a one of its main features. This same holds in particular for pervasive systems, where *instability* is one of the main issues, as well as for mobile systems, with any sort of mobility: physical mobility of users and devices; users who change their computing device while using applications; service instances migrating from machine to machine, as in a cloud-based environment.

The need for uncoupling mandates for *stateless interaction* in LPaaS. Thus, for instance, both LPaaS clients and service instances can freely move with no concerns for requests tracking and identity/location bookeeping.

In order to counterbalance the effect of statelessness on the typical LP user-engine interaction, LPaaS also provide clients with the ability to ask for more than one solution at a time, and even all of them, with a single request. Nevertheless, LPaaS also makes it possible to obtain a *stream of solutions* from the resolution process, rather than a single solution at a time in an individual interaction session, to better meet the needs of fast-paced dynamic scenarios in which clients want to be constantly updated by the LP service about some situation.

Accordingly, LPaaS provide clients with the means to obtain both stateless and stateful client-server interaction:

stateful once the logic theory to consider is settled, and the goal stated, the client should be able to ask for any amount of solutions, possibly iteratively, possibly at different times and from different places, with the service being responsible to guarantee consistency and validity of solutions by keeping track of the related interaction sessions with the same client;

stateless in this case, no session state is tracked by the server, so each client request should contain all the necessary (state) information.

It is worth highlighting that nothing prevents the service from being stateful and stateless simultaneously, because the LP server can manage multiple kinds of requests concurrently: of course, each client request in LPaaS is either stateful or stateless.

3.2.2 Microservices as Enablers

As mentioned above, SOA the standard approach for distributed system engineering: so, LPaaS adopts the *Software as a Service* (SaaS) architecture as its architectural reference [Cus10].

Accordingly, information technology resources are conceived as continuously-provided services: SaaS applications are supposed to be available 24/7, scale up & down elastically, support resiliency to changes (i.e., in the form of suitable fault-tolerance mechanisms), provide a responsive user experience on all popular devices, and require neither user installation nor application updates.

In particular, LP services in LPaaS can be fruitfully interpreted as *microservices* [Fam15]. Microservices are a recent architectural style for SaaS applications promoting usage of self-contained units of functionally with loosely-coupled dependencies on other services: as such, they can be designed, developed, tested, and released independently. Thanks to their features, microservices are deserving increasing attention also in the industry – pretty much like SOA in the mid 2000s – where fast and easy deployment, fine-grained scalability, modularity, and overall agility are particularly appreciated [Ric16].

Technically speaking, microservices are designed to expose their functionality through standardised network-addressable APIs and data contracts, making it possible to choose the programming language, operating system, and data store that best fits the service needs and the developers' skills set, without worrying about interoperability. Microservices should also be dynamically *configurable*, possibly in different forms and with different

Table 3.1: *LPaaS Configurator Interface*

`setConfiguration(+ConfigurationList)`
`getConfiguration(-ConfigurationList)`
`resetConfiguration()`
`setTheory(+Theory)`
`getTheory(-Theory)`
`setGoals(+GoalList)`
`getGoals(-GoalList)`

configuration levels. Of course, actual support to interoperability requires multiple levels of standardisation: to this end, LPaaS defines its own *interfaces* for both *configuration* and *exploitation*, while relying on widely adopted standards as far as the representation formats (i.e., [JSO17]) and interaction protocols (i.e. REST over HTTP, or [MQT17]) are concerned.

3.3 The Service

Following the reference architecture above, designing LPaaS amounts first of all at defining the Configurator Interface and the Client Interface—as in Figure 3.1.

Generally speaking, the LP service should support *(i) observational methods* to provide configuration and contextual information about the service, *(ii) usage methods* to trigger computations and reasoning, as well as to ask for solutions, and *(iii) configuration methods* to allow the configurator to set the LP service configuration.

Observational methods make it possible to query the service about its configuration (stateful/stateless and static/dynamic), the state of the knowledge base, and the admissible goals: as such, they belong to the Client Interface, but can be made available also in the Configurator Interface for convenience. Usage methods, instead, belong uniquely to the Client Interface: they allow clients to ask for one or more solutions—one solution, n solutions, or all solutions available, for stateful or stateless requests as well. Configurator methods belong uniquely to the Configurator Interface, and are intended to set the service configuration, knowledge base nature, and admissible goals.

3.3.1 Service Interfaces

Adopting the Prolog notation for input/output [DDC96], the actual Configurator methods are detailed in Table 3.1, while the Client Interface is detailed in Table 3.2. Since the first is rather self-explanatory, we focus on the Client Interface.

The first thing worth noting is that usage predicates are slightly different for stateless or stateful requests: in the former case, the `solve` operation is conceptually atomic and self-contained – `Goal` is always one of its arguments –, whereas in the latter case it is up to the server to keep track of the request state, so the goal is to be set only once before the first `solve` request is issued.

The second key aspect is the threefold impact of *time awareness*: regardless of whether the server is either computing or idle, time flows, so predicates must be time-sensitive:

- `solve` predicates can also contain a `Timeout` parameter (server time) for the resolution, so as to avoid blocking the server indefinitely: if the resolution process does not complete within the given time, the request is cancelled, and a negative response is returned;
- for *stateful requests*, the client could also ask for a *stream* of solutions, which is particularly useful in IoT scenarios exploiting sensor devices, or monitoring processes: to this end, `solve` takes a `time` argument (server time), meaning that each new solution should be returned not faster than every `time` milliseconds;
- when the KB is *dynamic*, all predicates take an additional `Timestamp` argument, meaning that each theory has a *time-bounded validity*: this feature can be used during the proof of a goal to ensure that only the clauses valid at the given `Timestamp` are taken into account in that resolution process.

For the sake of convenience, `solveAfter` methods provide for mimicking the LP stateful interaction on a stateless request channel, fast-forwarding to the `N+1` solution `AfterN`.

Finally, the `reset` primitive resets the resolution process, with no need to reconfigure the service (i.e., re-select the goal); in contrast, the `close` primitive actually closes the communication with the server, so the goal must be re-set before re-querying the server.

3.3.2 Computational Model

The computational model of the service is depicted by the Finite State Machine (FSM) in Figure 3.3, made of four states:

- *ready* (initial state), where the service is started and the engine is configured;
- *run*, where the service is undergoing some resolution process triggered by queries;
- *pause*, representing the temporary suspension of computations;
- *no goal selected* (final state), when the client connection is closed.

In the *ready* state, the service can be queried about its properties and a new goal can be set, thus defining a new resolution process. When a new query is submitted, the service

Table 3.2: *LPaaS Client Interface*

STATIC KNOWLEDGE BASE	
Stateless	**Stateful**
getServiceConfiguration(-ConfigList)	
getTheory(-Theory)	
getGoals(-GoalList)	
isGoal(+Goal)	
	setGoal(template(+Template))
	setGoal(index(+Index))
solve(+Goal, -Solution)	solve(-Solution)
solveN(+Goal, +NSol, -SolutionList)	solveN(+N, -SolutionList)
solveAll(+Goal, -SolutionList)	solveAll(-SolutionList)
solve(+Goal, -Solution, within(+Time))	solve(-Solution, within(+Time))
solveN(+Goal, +NSol, -SolutionList, within(+Time))	solveN(+NSol, -SolutionList, within(+Time))
solveAll(+Goal, -SolutionList, within(+Time))	solveAll(-SolutionList, within(+Time))
solveAfter(+Goal, +AfterN, -Solution)	
solveNAfter(+Goal, +AfterN, +NSol, -SolutionList)	
solveAllAfter(+Goal, +AfterN, -SolutionList)	
	solve(-Solution, every(@Time))
	solveN(+N, -SolutionList, every(@Time))
	solveAll(-SolutionList, every(@Time))
	pause()
	resume()
reset()	
close()	
DYNAMIC KNOWLEDGE BASE	
Stateless	**Stateful**
getServiceConfiguration(-ConfigList)	
getTheory(-Theory, ?Timestamp)	
getGoals(-GoalList)	
isGoal(+Goal)	
	setGoal(template(+Template))
	setGoal(index(+Index))
solve(+Goal, -Solution, ?Timestamp)	solve(-Solution, ?Timestamp)
solveN(+Goal, +NSol, -SList, ?TimeStamp)	solveN(+N, -SolutionList, ?TimeStamp)
solveAll(+Goal, -SList, ?TimeStamp)	solveAll(-SolutionList, ?TimeStamp)
solve(+Goal, -Solution, within(+Time), ?TimeStamp)	solve(-Solution, within(+Time), ?TimeStamp)
solveN(+Goal, +NSol, -SList, within(+Time), ?TimeStamp)	solveN(+NSol, -SList, within(+Time), ?TimeStamp)
solveAll(+Goal, -SList, within(+Time), ?TimeStamp)	solveAll(-SList, within(+Time), ?TimeStamp)
solveAfter(+Goal, +AfterN, -Solution, ?TimeStamp)	
solveNAfter(+Goal, +AfterN, +NSol, -SList, ?TimeStamp)	
solveAllAfter(+Goal, +AfterN, -SList, ?TimeStamp)	
	solve(-Solution, every(@Time), ?TimeStamp)
	solveN(+N, -SList, every(@Time), ?TimeStamp)
	solveAll(-SList, every(@Time), ?TimeStamp)
	pause()
	resume()
reset()	
close()	

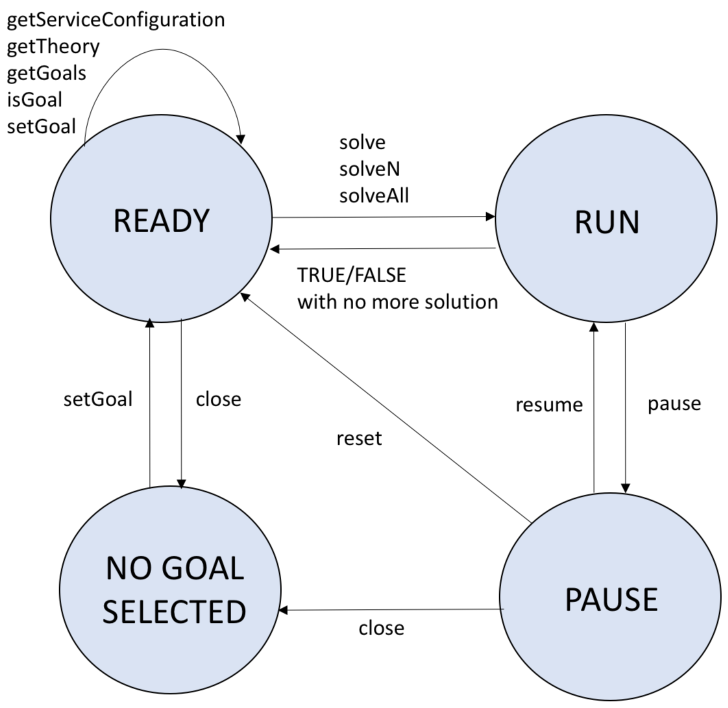


Figure 3.3: *The LPaaS Finite State Machine*

moves to the *run* state, indicating that a resolution process is taking place. Computation may then be paused several times, causing the service to move back and forth from the *pause* state: from there, resolution can also be reset (coming back to the initial state), or closed (moving to state *no goal selected*).

3.4 The LPaaS Technology

In this section, we present two different prototype implementations, as a Web Service and as an agent in a Multi Agent System, both built on top of the tuProlog system, which provides the required interoperability and customisation. We showcase the LPaaS potential through two case studies designed as a simplification of the motivating scenarios.

3.4.1 LPaaS as a RESTful Web Service

In order to test the effectiveness of the proposed architecture, we implement a first prototype of LPaaS as a RESTful web service (WS) [FT02]: we reuse and adapt patterns commonly used for the REST architectural style, and introduce a novel architecture supporting embedding Prolog engines into WS. Figure 3.4 shows the general architecture focussing on the server side and its components (access interfaces, Prolog engine, and

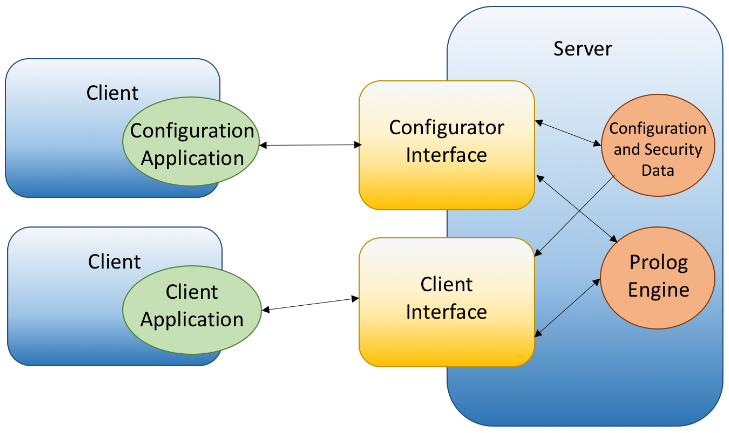


Figure 3.4: *The LPaaS RESTful WS.*

data store), as well as some exemplary client applications interacting via HTTP requests and JSON objects.

The server-side inner architecture (Figure 3.5) is composed by three logical units: the *interface* layer, the *business logic* layer, and the *data store* layer. The interface layer encapsulates the Configurator and Client Interfaces. The Business Logic wraps the Prolog engine with the aim of managing incoming requests consistently. The Data Layer is responsible for managing the data store tracking, i.e., all the configuration options necessary to restore the service in case of unpredictable shutdown (i.e., operating parameters and security metadata such as clients' role, username, password, . . .).

Since these data are expected to be limited in size for most scenarios, we choose to keep them in the server application so as to offer a light-weight, self-contained service: however, they could be easily moved to a separate persistence layer on, i.e., an external DB application, if necessary.

The server implementation is built by exploiting a number of technologies commonly available in the field: in particular, the Business Logic is built using the J2EE framework [J2E17], exploiting EJB [EJB17], whereas the database interaction is implemented on top of JPA [Jav17].

The Prolog engine is implemented on top of the tuProlog system [DOR01], which provides not only a light-weight engine, particularly well-suited for this kind of applications, but also a multi-paradigm and multi-language working environment, paving the way towards further forms of interaction and expressiveness. Since version 3.2, tuProlog also natively supports JSON serialisation, ensuring the interoperability as required by a WS. The tuProlog engine, distributed as a Java JAR (or, as Microsoft .NET DLL, or, as Android app), is easily deployable and exploitable by applications as a *library service*—that is, from a software engineering standpoint, a suitably-encapsulated collection of related functionalities.

The service interfaces exploit the EJB architecture, but can also be accessed as RESTful WS, realised using JAX-RS Java Standard (Jersey) [Jer17]. Security is based on jose.4.j [jos17], an open source (Apache 2.0) implementation of JWT and the JOSE specification suite [jos17]. The application is deployed using the Payara Application Server [Pay17], a Glassfish open source fork, and its source code is freely available on Bitbucket [tup01].

3.4.2 LPaaS RESTful Example Application

A testbed scenario, discussed in depth in [CDMO17], let us consider a Smart Bathroom to monitor physiological functions so as to deduce symptoms and diseases, and properly alert the user. Sensors collect data and undertake reasoning based on LPaaS provided by tuProlog, to come up with solutions made available to the user through a dedicated Android application. The Smart Bathroom system is composed of three different tuProlog-enabled LPaaS services processing data collected by

- toilet sensors analysing biological products, such as temperature, volume or glucose sensors like in [RTA14] (Toilet Server)

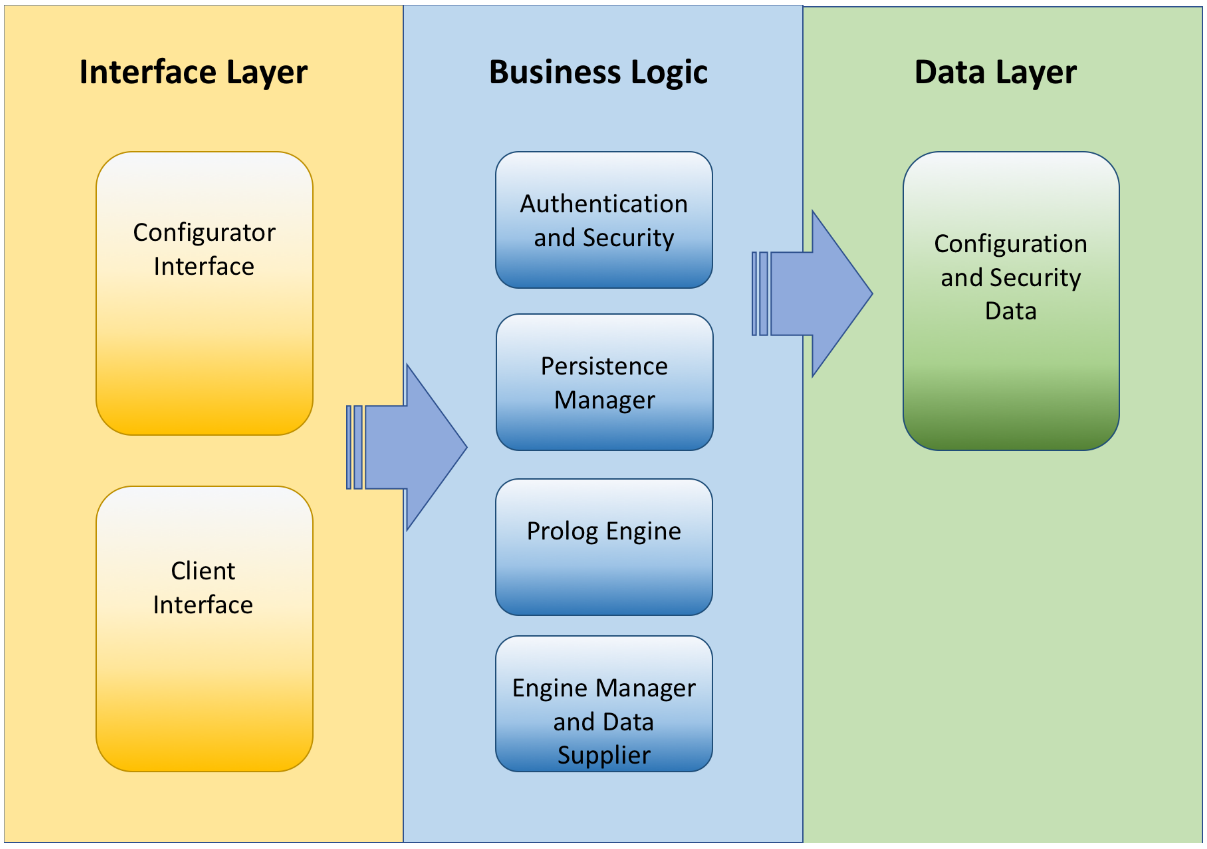
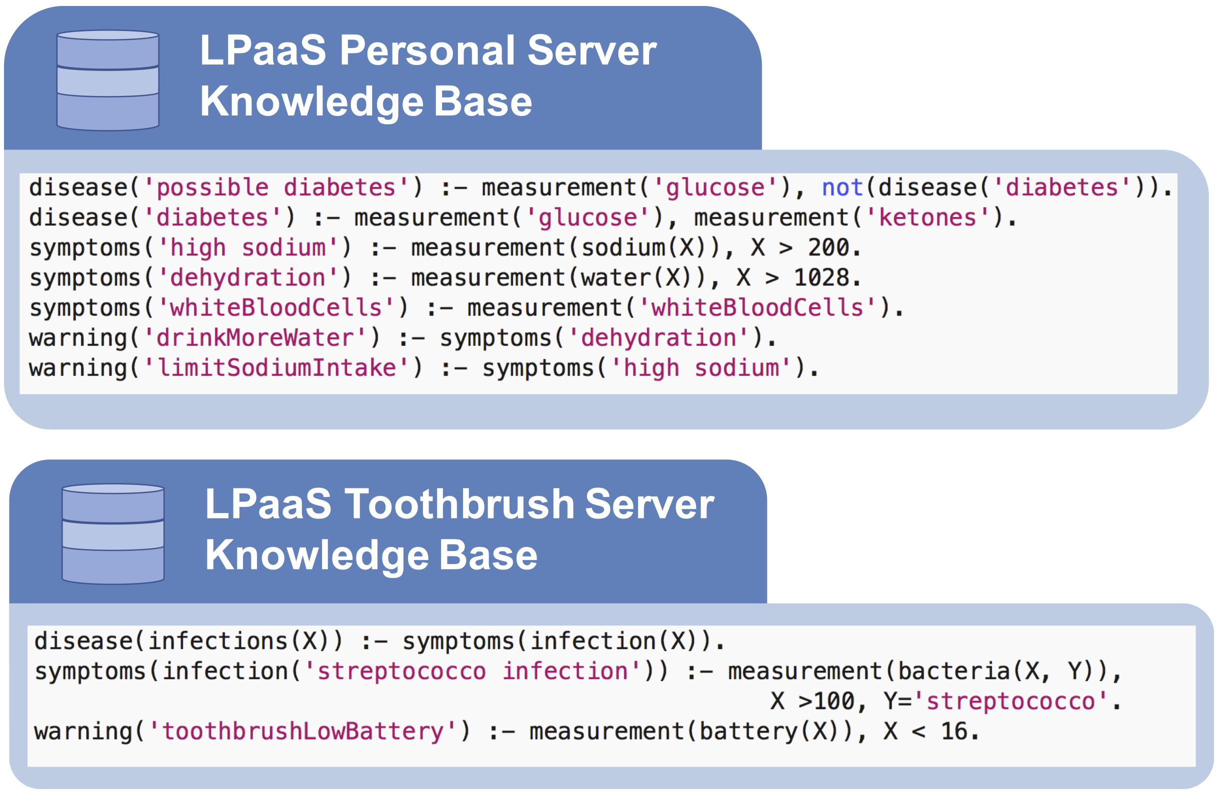


Figure 3.5: *The LPaaS WS server architecture.*


LPaaS Personal Server Knowledge Base
LPaaS Toothbrush Server Knowledge Base

```
disease('possible diabetes') :- measurement('glucose'), not(disease('diabetes')).
disease('diabetes') :- measurement('glucose'), measurement('ketones').
symptoms('high sodium') :- measurement(sodium(X)), X > 200.
symptoms('dehydration') :- measurement(water(X)), X > 1028.
symptoms('whiteBloodCells') :- measurement('whiteBloodCells').
warning('drinkMoreWater') :- symptoms('dehydration').
warning('limitSodiumIntake') :- symptoms('high sodium').
```

```
disease(infections(X)) :- symptoms(infection(X)).
symptoms(infection('streptococco infection')) :- measurement(bacteria(X, Y)),
                                                 X >100, Y='streptococco'.
warning('toothbrushLowBattery') :- measurement(battery(X)), X < 16.
```

Figure 3.6: *KB extraction of two different LPaaS server: namely, LPaaS Personal Server and LPaaS Toothbrush Server.*

- nano sensors integrated into the toothbrush (Toothbrush Server)
- ultrasonic bathtubs, pressure sensing toilet seats and other devices to monitor people's cardiovascular health (Personal Server)

Collected data may trigger different alerts: urgent ones, such as presence of Streptococcus infection, positive Diabetes Tests, etc. and normal ones, such as the need to drink more water, recharge batteries, and so on. An excerpt of the knowledge base of the services is shown in Figure 3.6.

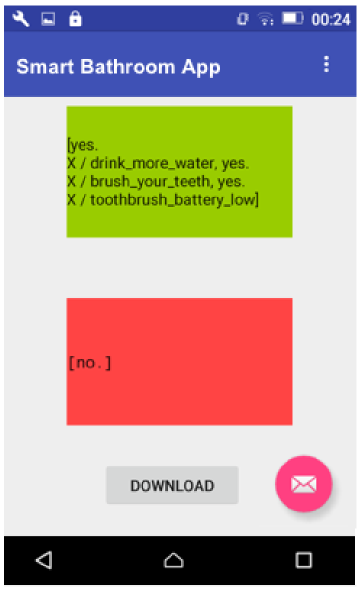
The system is built on the following hardware configuration:

- Toilet Server: Raspberry Pi 3 (Ubuntu Mate Arm)
- Toothbrush Server: Lubuntu laptop
- Personal Server: Windows 10 laptop
- Client 1: Lenovo A10 tablet with Android 5.0.1

- Client 2: Windows 10 laptop running a desktop application.

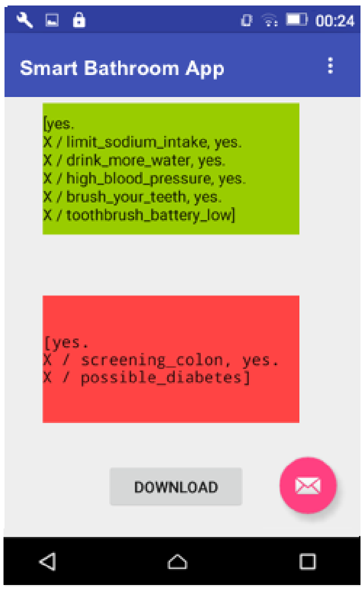
Currently, all the data collected by sensors are simulated: Figure 3.7 shows some screenshots of the Android application. Urgent messages are in red boxes, minor warnings in green boxes.

Despite its simplicity, the case study means to show the potential of the LPaaS approach: local sensors can perform *situated reasoning*, applying their *local knowledge* to aggregate the raw data and synthesise higher-level information. Such higher-order data can then enable the creation of new computing services that autonomously respond to a user, and provide more accurate predictions based on situatedness—in this case, provided by the Android application.



```
[yes.
X / drink_more_water, yes.
X / brush_your_teeth, yes.
X / toothbrush_battery_low]
```

```
[no.]
```



```
[yes.
X / limit_sodium_intake, yes.
X / drink_more_water, yes.
X / high_blood_pressure, yes.
X / brush_your_teeth, yes.
X / toothbrush_battery_low]
```

```
[yes.
X / screening_colon, yes.
X / possible_diabetes]
```

Figure 3.7: *The Android application exploiting LPaaS: non-urgent messages are shown in green, urgent ones in red. The left screenshot shows three non urgent messages (drink more water, brush your teeth, and toothbrush battery low), while the right one shows two non-urgent (limit sodium intake, high blood pressure) and two urgent messages (the possibility of diabetes, and the suggestion of a colon screening).*

3.4.3 LPaaS and Multi-Agent Systems

In this section we discuss how LPaaS can fit a Multi-Agent System (MAS), with the twofold aim of showing *why* merging LPaaS and MAS could be useful, especially in the IoT landscape, and *how* LPaaS and MAS could be successfully integrated.

Motivation

MAS for IoT. Agent-oriented engineering and MAS have been already recognised as a promising way of developing IoT applications and Cyber-Physical Systems (CPS), since they are well suited for supporting *decentralised*, *loosely-coupled* and *highly dynamic*, *heterogeneous* and *open* systems, in which components should *cooperate* opportunistically [CMS$^+$17, AK15, ZO04]. They also offer a higher level of abstraction to system designers and developers than, i.e., RESTful approaches, as they replace low-level notions such as HTTP requests / responses with messages and interaction protocols [ON98].

Adopting the thing-oriented definition of IoT – that is, the so-called Smart Object (SO) IoT vision in which SOs are the basic IoT building blocks –, it is quite natural to map the sensing and actuating capabilities of SOs onto the perception and action capabilities of *situated agents* [HBKR10]. Also, SOs are meant to be autonomous in acting on behalf of their owner in the most common everyday activities such as, for a smart home scenario, adjusting temperature, tuning lights intensity, lock the doors, and so on. Not by chance, *autonomy* is also the core feature of agents [HCF03].

Another straigthforward mapping may be drawn between the need for cooperation amongst SOs in complex IoT scenarios and the *social dimension* of agency [Cas98]—and, consequently, of MAS. In fact, SOs can be expected to interact with each other in order to perform even simple tasks, such as those related to situation recognition, and the usual means to do so in current IoT practice is either *(i)* to let the Cloud handle dependencies between tasks, for instance by monitoring a given sensor perceptions to trigger a given actuator when a threshold is met, or *(ii)* to exchange very simple messages (i.e. JSON structures) in a peer-to-peer way. Agents, instead, are naturally capable of diverse forms of social interactions, by exchanging rich messages in compliance with well-defined protocols having a clear and well-understood semantics—i.e. FIPA protocols and ACL messages [ON98].

Finally, featuring a *goal-oriented/driven* behaviour [Cas12], agents can plan and act based on the specific contingencies of the environment in which they operate. This deeply contrasts the imperative way of commanding SOs in current IoT practice, where actuator devices are usually only able to react to precise and direct instructions about what/how *to do*, not what *to achieve*, as a more declarative approach would suggest.

In spite of the above benefits, a relevant issue may hinder adoption of the agent abstraction, thus of MAS, in the IoT landscape: the computational limitations of SOs, which can be too severely resource-constrained to embed a full-fledged software agent. Here is where LPaaS comes into play, as explained in the next section.

LPaaS for MAS. Besides *autonomy*, *situatedness*, and *sociality*, agents may have other features that could map onto SOs: for instance, mobility – intended as code mobility – can be easily implemented even on resource-constrained devices, whereas *intelligence* – a hot topic in current IoT research – is considerably a more challenging issue. The fact is that providing a reasonable perception of intelligence for a given SO (agent) requires many different technologies, such as machine learning, common-sense reasoning, natural language processing, advanced situation recognition and context awareness—which are all typically computationally expensive *per se*, let alone in conjunction with the others. This is why the concept of LPaaS may actually improve the state of art in engineering intelligent IoT systems: with LPaaS, just the "required amount" of situated intelligence can be seamlessly spread where needed, and/or where the available resources are able to bear the computational effort, with no need to have a full-fledged intelligent agent embedded in every SO.

In this new perspective, whenever local intelligence cannot be available for any reason – i.e. memory constraints hindering the opportunity to have a local KB, CPU constraints limiting efficiency of reasoning, etc. – a given agent (SO) may simply request to another, "more intelligent" one, to perform some inferences on its behalf. Moreover, the LPaaS functionality may also be charged upon the infrastructure, instead of the agents. In this scenario, agents are always computationally efficient and responsive, since they delegate reasoning-related tasks – such as situation recognition, planning, inference of novel information, etc. – to dedicated infrastructural services—either hosted in the Cloud, as it currently happens for most IoT platforms, or spread amongst a distributed set of devices working as gateways for SOs.

As a last remark, traditional LP has well proven valid over time both as a knowledge representation language and as an inference platform for rational agents. Logic agents may interact with an external environment by means of a suitably defined observe–think–act cycle. Significant attempts have been made to integrate rationality with reactivity and proactivity in logic programming [KS96, DST98, KS99]: the re-interpretation of LP under the LPaaS approach in MAS could be seen as the evolution of these research threads for modern pervasive and distributed systems.

Fitting LPaaS in MAS

Engineering a MAS generally requires three orthogonal yet complementary dimensions [ORV08] to be considered: the *agents* dimension, where the internal structure of agents is designed and their behaviour programmed; the *social* dimension, where the focus is on the space of interaction [OOR04], thus in designing how agents interact; the *environment* dimension, where the representation of anything in the physical or computational world relevant to the MAS itself lives. Integrating LPaaS with MAS thus requires first to carefully decide where the integration should take place—that is, along which dimension, and involving which abstractions.

agent Integrating along the agent dimension is probably the most natural way to proceed, as it directly injects intelligence more closely to the agents' business logic, or even deeply in their inner reasoning workflow—likewise for BDI agents [Rao96]. Nevertheless, it is probably also the least suitable for the average IoT scenario, where devices can be severely resource-constrained, hence unable to host a software agent together with a LP engine. LPaaS is conceived precisely to enable this kind of integration with no need to host the LPaaS engine within the agent itself: the engine can in fact be hosted anywhere, and made accessible to agents through a dedicated service layer—be it implemented according to REST, as exemplified in Section 3.4.1, or to any other architectural style / technology.

society Integrating LPaaS and MAS along the social dimension amounts to embed LP functionalities in the *coordination artefacts* devoted to manage the interaction space [ORV06]. This approach is similar to the one adopted in TuCSoN [OZ99a] with the ReSpecT coordination language [Omi07], where coordination rules enacted by enhanced tuple spaces [OD01b] are expressed in a Prolog-like language (actually interpreted by a tuProlog engine, the same used for LPaaS).

environment The last alternative is to consider LPaaS services as part of the MAS environment. Usually this means deploying LPaaS as a *middleware* service, provided by the infrastructure hosting agents and enabling them to access the network and devices' capabilities. This approach could be implemented by exploiting LPaaS as a RESTful WS, as described in Section 3.4.1, and is complementary to the first one, in the case agents do not embed an LP inference engine but exploit LPaaS opportunistically.

The three aforementioned approaches are not to be taken as mutually exclusive: in fact, our choice in the prototype system is to exploit the first and last one together, leaving MAS designers free to choose whether they mean to embed intelligence within agents, or, instead, to have it provided as an infrastructural service.

Figure 3.8 illustrates the model of the LPaaS approach depicting the whole picture in the hybrid case where *(1)* some agents are kept more lightweight and rely on infrastructural services (or other more "intelligent" agents) to get LPaaS functionalities, *(2)* some agents embed the LPaaS functionalities, and *(3)* some LP functionalities are embedded in some services provided by the middleware (namely by the containers).

The traditional MAS architecture is enriched with the notion of LPaaS agent / service, which allows for situated reasoning on locally-available data by design. In this vision, agents can be split in two groups: *local* agents and *global* agents. The former includes all the agents embedded in sensor and actuator devices, and in charge of generating the *local knowledge*: they represent the local view of the IoT system. The latter group includes agents with a higher-level view of the system, not necessarily embedded in SO devices, which act and coordinate their activities to properly pursue the system's goal. In the service layer, LPaaS and other typical middleware / application services are supplied. These services can be provided by the devices located in the physical world, by the MAS agents, or by dedicated infrastructural components.

Figure 3.9 (left) illustrates in detail the case where SO agents embed the LPaaS service (namely, case 2), and are therefore able to both perform their own reasoning and offer their

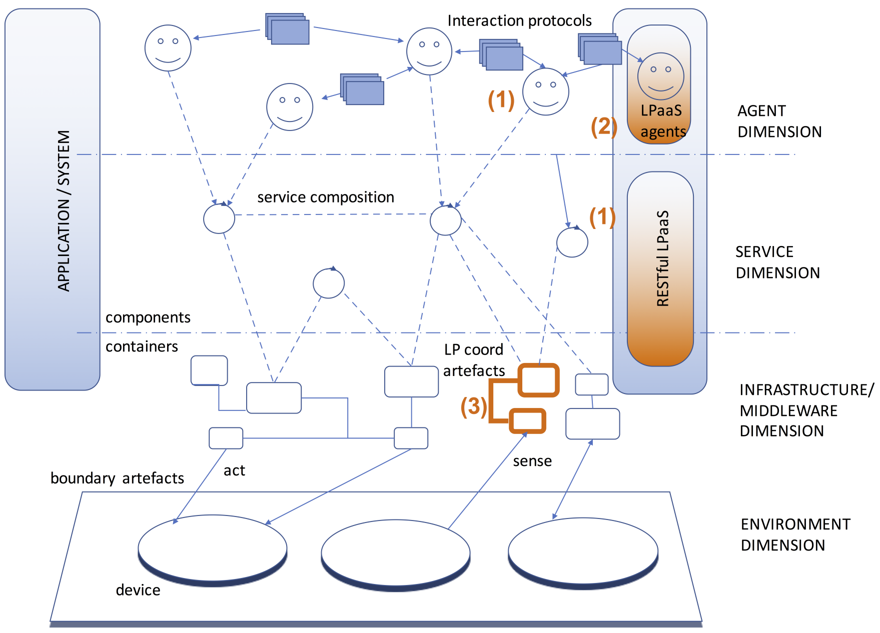


Figure 3.8: Overview of a LPaaS multi-agent system. At the bottom layer, the physical / computational environment lives, with boundary artefacts *[ORV06] taking care of its representation and interactions with the rest of the MAS. Then, typically, some middleware infrastructure provides common API and services to application-level software – i.e. the containers where service components live – there including the* coordination artefacts *[ORV06] governing the interaction space. Finally, on top of the middleware, the application / system as a whole lives, in LPaaS MAS view as a mixture of services – possibly RESTful, as for LPaaS as a WS – and agents.*

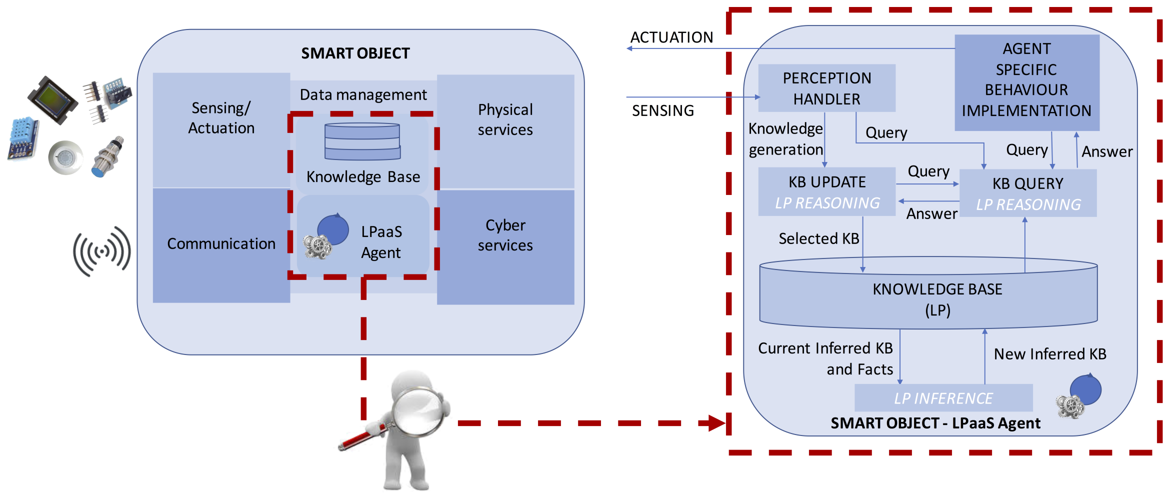


Figure 3.9: The Smart Object as an LPaaS Agent: typical SO conceptual architecture enriched with LPaaS service (left) and inner architecture (right).

capabilities to others. In this case, each SO has a representative autonomous software agent, which is capable of monitoring the state of the device, make decisions on behalf of the device, and discover and exploit external help if necessary.

Revisiting Agents' Inner Architecture

Figure 3.9 (right) shows the inner architecture of a SO modelled as an LPaaS agent:

- The *Perception Handler* takes care of measurements coming from sensors. The new data feed the knowledge base of the SO, may alter existing knowledge, and contribute to the inference of novel knowledge
- The *KB Update* component evaluates incoming data, and updates the knowledge base accordingly. A trivial implementation of the LPaaS agent can simply insert novel information with no modifications or restrictions (actually acting as an information repository), whereas a more sophisticated one interacts with the *KB Query* component to perform consistent updates of the KB (actually supporting the interpretation of the KB as a logic theory of the local world)
- The *KB Query* component receives queries from the agent specific behaviour implementation, from the Perception Handler, and from the KB Update component. In the first case, it supports the agent's internal decision-making; in the second, it helps the KB to remain consistent while updating itself in reaction to external stimuli. In the latter case, it helps in selecting which information will be inserted in the KB
- The *Knowledge Base* is a logic theory. In principle, any modification of the surrounding environment perceived by sensors can produce an update
- The *LP Inference* component performs deduction from the existing KB. Its inputs is the current KB, which is a mixture of background knowledge, measured metrics, and deduced facts; its output is the novel inferred knowledge.

On LPaaS Agents' Lifecycle

To realise the LPaaS MAS architecture, a *container-component* programming model can be adopted: the LPaaS agent is a component living inside a container that manages its life cycle. The container-component model simplifies the configuration and usage of the LPaaS inferential engine in a distributed system, by *separating concerns*: on the one hand, the component is the one responsible for the LPaaS inferential engine and its business logic; on the other, the container is a portion of the middleware that manages the number of instances, configuration, and life cycle of the handled components.

Accordingly, an LPaaS agent is handled by an LPaaS container which manages the service's core. The agent is characterised by a cyclic behaviour: at each iteration, it receives a service request from a client, synthesises the response, and communicates the operation result. Figure 3.10 shows the operations performed by the container for creating and configuring an LPaaS component: the container, after loading the component's metadata, creates the component, configures the inferential engine, and runs the automatic methods (for self-configuration).

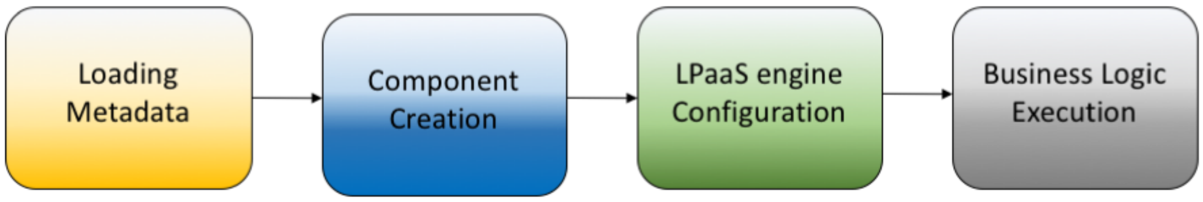


Figure 3.10: *The LPaaS container in a MAS.*

3.4.4 LPaaS MAS Prototype: LPaaS in Jade

The proposed approach has been implemented on top of the Jade middleware [BCG07, JAD], which facilitates the development of interoperable, open, and heterogeneous multi-agent systems by relying on the FIPA standard [ON98], and exploiting tuProlog [DOR01, tup01] as the LPaaS Prolog engine. We choose tuProlog because of its peculiar blend of imperative, object.oriented, and logic programming styles: apart from being Java-based, light-weight, and easy deployable, it also enables and promotes a *multi-paradigm* programming style, where the Prolog code can invoke Java code and viceversa, yet keeping the two computational models clearly separate [DOR05].

Following Jade approach to openness, the LPaaS agent must register with the Jade Directory Facilitator (DF) – Jade yellow pages for services offered by agents in a Jade MAS – by providing a logical identifier and indicating the sort of service offered (i.e., LPaaS). In this way, clients can dynamically perform a discovery operation and identify which LPaaS agents live in the system—since many may provide the LPaaS service simultaneously, i.e. for resiliency and performance reasons. In its turn, a client wishing to use a LPaaS service must, as a first action, perform a discovery via the DF, requesting either the list of all agents offering LPaaS services, or a specific service, given its logical identifier.

The communication between Jade agents occurs via ACL (Agent Communication Language) messages [FIP02], that is, well-structured messages with a clear semantics and interoperable encoding. The request ACL message for an LPaaS service is always a FIPA Request, and should contain both the logical identifier of the LPaaS agent and the identifier of the operation to be run (a string that uniquely identifies a method). The request message may also contain the goal to demonstrate, possibly the number of solutions to be scanned, and the maximum service running time (i.e., timeout), depending on the nature of the LPaaS configuration (i.e. stateless vs. stateful, dynamic vs. static). The response ACL message is always a FIPA *Inform* [JAD], notifying the customer of the service result: the response message contains either the requested solutions or an error message if the service could not be run.

The client agent, who obtained the Jade AID (Agent IDentifier) of one or more agents from the DF as a result of the discovery phase, sends an LPaaS Request ACL message to the selected agent: in turn, the LPaaS agent replies with an LPaaS response message, which contains the service outcome (Figure 3.11). Interaction always adheres to the request-response pattern: the LPaaS agent is supposed to reply to the client in all cases, possibly with a failure message in case of errors.

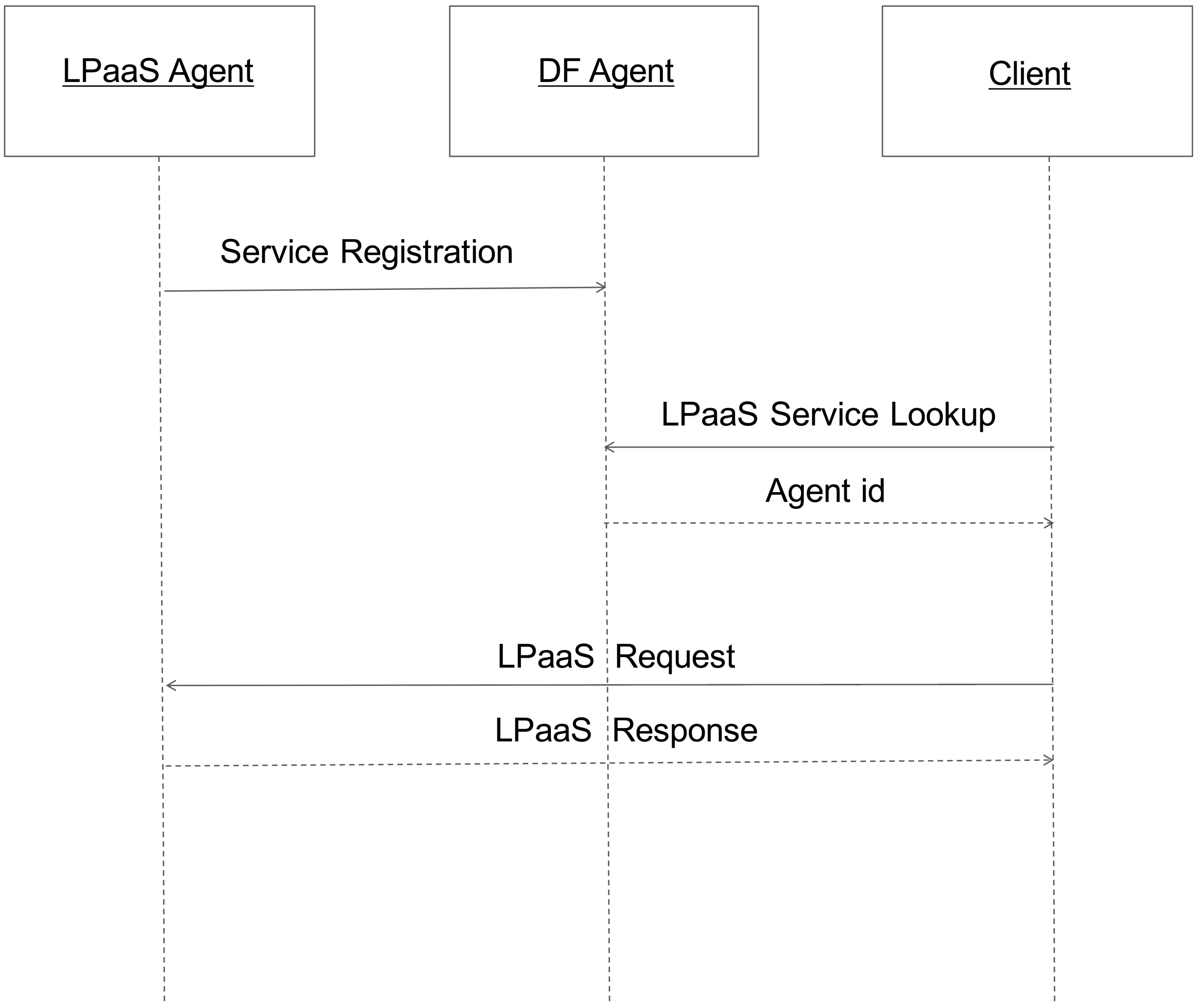


Figure 3.11: *The LPaaS Service-DF-Client interaction.*

The reception of the message may be blocking or not blocking, both for the LPaaS agent and the client, depending on the configuration parameter: blocking mode is the default for the LPaaS agent (so, with no service requests the agent is suspended), while the client fully depends on the application logic.

As far as security is concerned, two aspects are currently supported: *(i)* authentication and confidentiality of the communication; *(ii)* identification of client's permissions to access the service. The first is ensured using JADE-S (Secure JADE) [PRT01]: each LPaaS message is signed and encrypted. In particular, when an LPaaS agent receives a service request message, it first checks whether the message is signed by a known agent (via JADE-S) and only in this case proceeds by decrypting the message; otherwise, an error message is sent back. The second level implies that LPaaS agents can distinguish privileged, configurator agents which can start, stop, and reconfigure (admissible goals, kb, ...) the service.

Example Application

As a first testbed for LPaaS in JADE, we implemented a Smart Kitchen IoT scenario. Four IoT devices – namely, a fridge, a pantry, a mixer and an oven – supply information to clients, exploiting the LPaaS approach, about the food supply and users' preferences.

The fridge and the pantry are capable of monitoring the quantity of food, and of collecting historical data on user's habits, i.e. the most commonly eaten food and preferred meals. The oven aims at supporting the user's cooking experience by relying on any available technology to identify and cook food. The user profile is supposed to include information about his/her dietary requirements. The mixer manages the recipe instructions, interacting with both the fridge – to check that the ingredients for the selected recipe are actually available – and with the oven—to check its ability to cook that food, and potentially synthesise the proper control instructions. Each device is supposed to have a limited computational capacity, such as that of a Raspberry Pi or of an Arduino board.

The application scenario requires that each device does not merely provide raw data, but is instead capable of producing higher-level knowledge and simultaneously of coordinating and collaborating with other entities in the system. In particular, each node (identified by a device) must maintain local knowledge about its status, be aware of the surrounding environment, and be able to communicate with the control device to share information about the kitchen state. Moreover, it should be possible to migrate the software from a device to another (e.g. if the device needs to be replaced) and to add / remove IoT devices to the system without shutting the system down first, i.e. in a "plug and play" fashion.

Assuming that all IoT and control devices are connected to a single home subnet, we choose to adopt a single JADE platform: the JADE Main Container is located on the controller node and is in charge of interacting / retrieving information from the smart kitchen devices. Each IoT device is designed as an LPaaS component, and is supposed

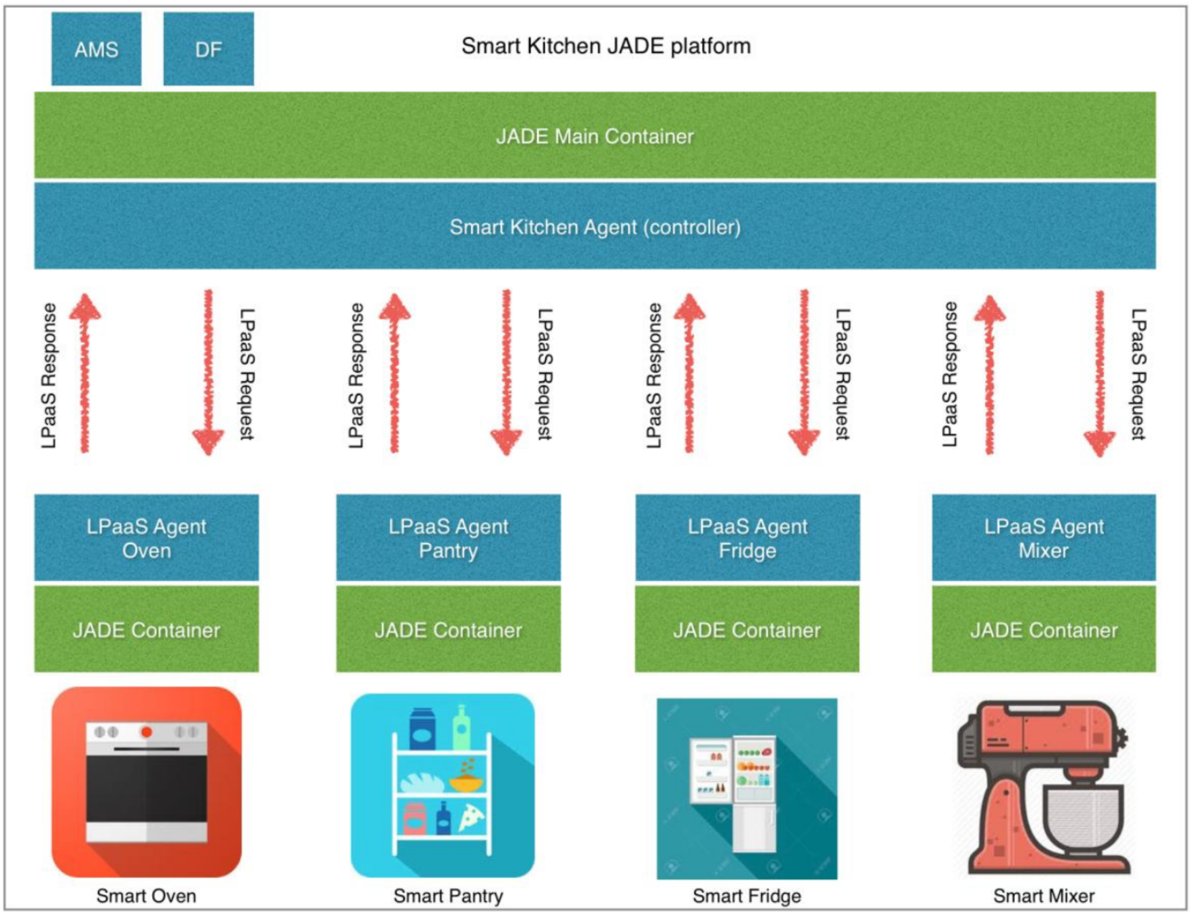


Figure 3.12: *The Smart Kitchen Architecture in a LPaaS* tu*Prolog in* JADE.

to manage the node knowledge base and expose the goals. In this case, the available goals make it possible to query devices about available food and the user's habits—e.g., trace the available products, quantities, expiration date for perishables, purchase price and retailer, origin, users' preferred products, etc.

Figure 3.12 gives an overview of the corresponding JADE system. The Smart Kitchen Agent is the *global* agent, in our terminology, in charge of ensuring a coherent behaviour of the overall system based on the overall knowledge gathered. The interaction between the Smart Kitchen Agent and the LPaaS *local* agents – that is, those responsible of gathering local information and providing situated reasoning – occurs via ACL messages: the Smart Kitchen can thus obtain high-level information from the devices, process it, and decide the action(s) to be taken. For instance, if a given kind of food in the fridge is running out, the Smart Kitchen Agent may place an online order.

Some screenshots from the current experimental prototype are shown in Figure 3.13.

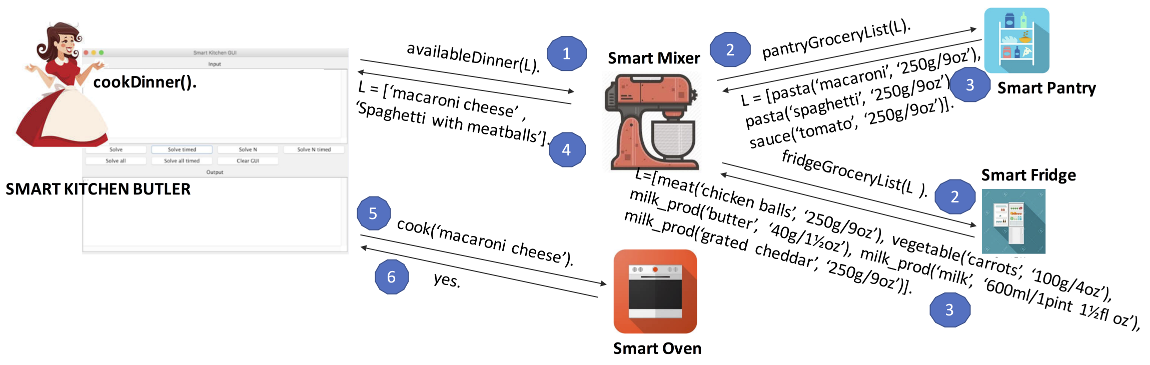

cookDinner().
SMART KITCHEN BUTLER
1 availableDinner(L).
4 L = ['macaroni cheese', 'Spaghetti with meatballs'].
Smart Mixer
2 pantryGroceryList(L).
3 L = [pasta('macaroni', '250g/9oz'), pasta('spaghetti', '250g/9oz'), sauce('tomato', '250g/9oz')].
Smart Pantry
2 fridgeGroceryList(L).
3 L=[meat('chicken balls', '250g/9oz'), vegetable('carrots', '100g/4oz'), milk_prod('butter', '40g/1½oz'), milk_prod('milk', '600ml/1pint 1½fl oz'), milk_prod('grated cheddar', '250g/9oz')].
Smart Fridge
5 cook('macaroni cheese').
6 yes.
Smart Oven

Figure 3.13: *The Smart Kitchen Prototype: example of possible inferences.*

In its simplicity, the example scenario outlined here showcases how easy it is to spread situated intelligence in a IoT deployment by merging LPaaS with MAS. The Smart Kitchen agent, for instance, needs not to bother tracking low-level data such as the amount and kind of perishables, recipes requirements, etc. altogether, so as to have all the system knowledge available and plan action accordingly. This is what typically happens in Cloud-based IoT deployments. In LPaaS MAS instead, the Smart Kitchen agent may directly ask to its peers the higher-level information it needs for decision making – with queries such as "May I start cooking a lasagna?" or "Does Lisa like broccoli?" – expecting an informative reply—instead of a raw measurement, such as "We are missing besciamella for lasagne".

3.4.5 Benefits & Open Issues

In this section we discuss the envisioned benefits of the LPaaS approach, both in itself and merged with MAS, especially considering the IoT landscape, and along with the open issues to be dealt with for fully realising LPaaS, and LPaaS in MAS.

The first benefit, discussed throughout the Chapter, is enabled by LPaaS alone: *ubiquitous intelligence* for pervasive scenarios. LPaaS enables system designers to distribute reasoning and inference capabilities amongst the components they have, and let them balance the computational requirements to best suit the deployment scenario at hand—for instance, embedding LPaaS in more powerful components and letting ask their services by need. To the best of our knowledge, this is something unprecedented in the current IoT landscape, where most approaches either assume a fully-distributed network of smart objects capable of performing general-purpose computations, or resort to a Cloud-based setting where the whole system intelligence resides on the Cloud side.

The second benefit naturally follows: *situated reasoning*. LPaaS enables reasoning and inferential processes to be context-aware w.r.t. the (possibly ever-changing) environment where the process takes place. For instance, a sensor augmented with LPaaS capabilities

– or, able to interact meaningfully with an LPaaS service – can reason on the locally-gathered data and provide to other system components not just raw measures, but high-level, inferred information about the sensed situation. In turn, this enables actuators to carry out a situated decision-making process, where the course of actions to undertake is the result of a situated planning (inference) process. It is worth noting that resorting mostly to locally available information reduces both the bandwidth consumption and the need for reliable communications between the distributed components, which are especially desirable features in IoT scenarios.

Furthermore, other benefits can be envisioned when coupling LPaaS with MAS: for instance, *goal-orientedness*. LPaaS agents may in fact exploit LPaaS to reason about their own goals, the plans and actions needed to achieve them, and the effects brought by—which is something only rational agents (such as BDI ones [Rao96]) usually do. Also, even non-LPaaS agents and non-agent components may do so, by simply interacting with the available LPaaS services. This is a simple and effective way to inject goal-orientedness in components of any kind, regardless of their inner architecture and implementation logic.

Lastly, when compared with the RESTful WS approach, the MAS-based one has some notable advantages: *(i)* complex interaction protocols built upon semantically rich messages (i.e. FIPA protocols on ACL messages); *(ii)* an interaction model particularly suitable for decentralised computations, based on the peer-to-peer model, yet not imposing a strict separation between client and server roles; and *(iii)* potential *mobility* of the service through the agent's own mobility.

In order to fully exploit the potential of the LPaaS approach, a few open issues are yet to be addressed.

For instance, deploying LPaaS in a real IoT scenario is likely to require integration with databases, possibly distributed, which should work as the distributed knowledge base of the system. Then, the issue of handling replication and consistency of data scattered in connected devices arise, in particular when a coherent, logical interpretation of the data in LP terms is required—that is, as a logic theory of the state of the world.

Related to this, strict integration with sensor devices is desirable, so as to have LPaaS always working the most up-to-date perception of the environment properties of interest for the application at hand. In this respect, devising out ways to automatically embed the process of gathering sensors' perceptions into the LPaaS working cycle could prove to be extremely useful in facilitating adoption of the LPaaS approach and embedding of LPaaS within devices.

On the opposite side of the IoT spectrum – that is, looking at actuator devices – deep integration with their operation API is welcome, so as to have the LPaaS distributed engine automatically command devices whenever some reasoning process results in the need of interacting with the physical world.

Another open issue is how to deal with situatedness in space, and mobility. Many modern applications would in fact benefit from having logic theories and inference processes somehow bound to spatial aspects of the application domain—similarly, and complementarily, to what LPaaS does with time. For instance, resolution of a query may consider also logic theories and LP engines in the neighbourhood of the queried one, and LPaaS engines may be able to move from node to node in search for better computational resources.

All the above issues are presently under consideration and will be subject of future research.

3.5 Remarks & Outlook

In this Chapter we presented the LPaaS approach for *distributed situated intelligence* as the natural evolution of LP in nowadays pervasive computing systems. We discussed its properties and its computational and architectural models in constant relationship and comparison with the notions and development of LP over the years.

The main advantages of applying an LP approach to pervasive systems can be summarised as

- the chance of writing declaratively complex rules that involve the context,
- the empowerment of designers in making provable statements about the expressive power and decidability of the context model, and
- the possibility of actually supporting light-weight reasoning and cooperation among distributed components.

Our service-based approach, in particular, *(i)* encourages representing and reasoning with situations using a declarative language, providing a high level of abstraction; *(ii)* supports the incremental construction of context-aware systems by providing modularity and separation of concerns; *(iii)* promotes the cooperation and interoperation among the different entities of a pervasive system; and *(iv)* enables reasoning over data streams, like those collected by sensors.

We also presented a first prototype implementation built on top of the tuProlog system, to demonstrate and test the effectiveness of the LPaaS approach. Our implementation is designed on the top of tuProlog, a light-weight, multi-platform, and multi-language engine that is well suited for the purpose. We discussed and implemented two different architectures: the first is based on the usual SOA infrastructure—namely, RESTful web services [FT02], while the second is based on multi-agent system [Fer99]. In addition, we discussed the integration of these different paradigm and solutions, highlighting the advantages of such a hybrid approach.

Of course, a number of enhancements, both to the model and the infrastructure, are still possible. From the model viewpoint, the LPaaS interface can be extended with specific space awareness methods to consider the space around either the client or the server, exploring the chance to opportunistically federate LP engines upon need as a form of dynamic service composition. Space-awareness and situatedness will be investigated, exploring the idea to opportunistically federate LP engines by need as a form of dynamic service composition. From the infrastructure viewpoint, we plan to focus on the design and implementation of a specialised LP-oriented middleware, dealing with heterogeneity of platforms as well as with distribution, life-cycle, interoperability, and coordination of multiple situated Prolog engines – possibly based on the existing tuProlog technology and TuCSoN middleware [OZ99a] – so as to explore the full potential of logic-based technologies in IoT scenarios and applications.

Part II

Dealing with Situatedness & Domain-specific Scenarios in Logic Programming

In this second part of this thesis, the Situatedness & Domain-specific issue in Logic Programming is addressed.

The vast majority of applications, nowadays, have to be designed taking into consideration the local situation in which entities are involved, that is, space, the people around, and the local activities, because they may affect the perception and the knowledge of the entity leading to different solutions in the deduction process. Indeed, context-awareness enables a system to take action automatically, reducing the burden of excessive user involvement, and providing proactive intelligent assistance. Logic programming is generally useful for such reasoning and for supporting the rule-based programming paradigm in context-aware applications. Dealing with situations, and therefore with the locality in which each entity is immersed, leads to the requirement of shaping the domain specificity. Of crucial importance, in this task, is the possibility to use different paradigms / languages to express the domain specific situation, and capturing the local model.

Accordingly, in this part of the thesis, the state of the art in the literature is discussed, focussed on the models and technologies which mostly influenced the thesis (Chapter 4); then, the LVLP model is presented as our contribution to enable LP to deal with the diversity of pervasive systems, where many heterogeneous, domain-specific computational models could benefit from the power of symbolic computation (Chapter 5).

Chapter 4

State of the Art

4.1 Logic Programming & Situatedness

Surveying the literature reveals a large number of diverse proposals pushing computational logic towards *distributed situated intelligence* [Par08] in pervasive systems—to exploit domain knowledge, understand local context, and share information in support of intelligent applications and services [CFJ03, Sma17]. There, systems are expected to respond intelligently to contextual information about the physical world acquired via sensors and information about the computational environment.

Many languages extension have been proposed in order to allow intelligent agents to interact with the environments and deal with specific situation, highlighting the benefits of LP for reasoning in pervasive systems. XLOG [FB98] is a hybrid programming environment where predicate logic is integrated into an object-oriented computational model, specially adequate for working with reactive agents to enable the principles of emergence and situatedness. Along this line, CIFF [EMS$^+$04] is a system implementing a novel extension of Fung and Kowalski's IFF abductive proof procedure [FK97] aimed at building intelligent agents that can construct plans and react to changes in the environment. The proposed solution improves more conventional abductive theories for planning by adding the possibility to interact with the environment, by observing environment properties as well as actions executed by other agents, thus enhancing agent situatedness.

Moreover, many researches exploit LP extensions to model context and situations. In the works by Ranganathan and Campbell [RC03] and Katsiri and Mycroft [KM03], FOL)is used for representing and reasoning with context, whereas Henricksen [HIR02] exploits FOL to describe and reason with situations. On the other hand, the above way does not adopt a modular approach or meta-reasoning as in [Lok04], where an extension of Prolog (LogicCAP) is presented: the notion of *situation program* is introduced, thus highlighting the primacy of the situation issue for building context-aware pervasive systems.

The need for a more specialised language support to solve problems in well-defined application domains has resurfaced constantly in the research over the years. Over time, the following solutions have been tried:

- Subroutine libraries contain subroutines that perform related tasks in well-defined domains like, for instance, differential equations, graphics, user-interfaces and databases. The subroutine library is the classical method for packaging reusable domain-knowledge [vDKV00].
- Object-oriented frameworks and component frameworks continue the idea of subroutine libraries. Classical libraries have a flat structure, and the application invokes the library. In object-oriented frameworks it is often the case that the framework is in control, and invokes methods provided by the application-specific code [JF88, FS97].
- A domain-specific language (DSL) is a small, usually declarative, language that offers expressive power focused on a particular problem domain. In many cases, DSL programs are translated to calls to a common sub-routine library and the DSL can be viewed as a means to hide the details of that library [Ken97].

Domain-specific languages are usually declarative. Consequently, they can be viewed as specification languages, as well as programming languages. Many DSLs are supported by a DSL compiler which generates applications from DSL programs. In this case, the DSL compiler is referred to as application generator in the literature [Cle88], and the DSL as application-specific language. Other DSLs, such as YACC [Ben86] or ASDL [WAKS97], are not aimed at programming (specifying) complete applications, but rather at generating libraries or components. Also, DSLs exist for which execution consists in generating documents (TEX), or pictures (PIC [Ben86]). A common term for DSLs geared towards building business data processing systems is 4th Generation Language (4GL).

4.2 Logic Programming & Labels

Orthogonally, Labelled Deductive Systems (LDS) have been proposed for providing logics from different families with a uniform presentation of their derivability relations and semantic entailments to deal with domain-specific situations [Gab96]. The main idea there is to provide a new unifying methodology, replacing the traditional view of logic, manipulating sets of formulas by the notion of structured families of labelled formulas. LDS is a unifying framework for the study of logics and of their interactions. In the LDS approach the basic units of logical derivation are not just formulae but labelled formulae, where the labels belong to a given *labelling algebra*. The derivation rules act on the labels as well as on the formulae, according to certain fixed rules of propagation. By virtue of the extra power of the labelling algebras, standard (classical or intuitionistic) proof systems can be extended to cover a wider territory without modifying their structure.

Detailed investigations have been undertaken to explore the benefits of using the LDS methodology to reformulate intuitionistic modal logics [Sym94] and substructural logics [BFR99, DGB99]. Specialised frameworks based on LDS have been also proposed

[AGR02, Bla00, Rus96]. Among the others, the Compiled Labelled Deductive Systems (CLDS) approach demonstrated how LDS techniques facilitate the reformulation and generalisation of a large class of modal logics and conditional logics [Rus96, BGLR02].

Along the same line, deductive object-oriented databases (DOODs) seek to combine the complementary benefits of the deductive and the object-oriented paradigms in the context of databases exploiting labels mechanism. Research into DOODs has been taking place for almost ten years, and a significant number of designs and implementations have been developed. It was at first felt that paradigms were incompatible, or that combining them would lead to unacceptable compromises, but in fact a number of proposals have been made that support both comprehensive deductive inference and rich object-oriented modelling facilities. DOODs are classified according to the language design strategy exploited during their development. There is more variety in the approaches taken to the development of DOODs than was the case with deductive relational databases (DRDBs), as in the latter case an agreed data model gave rise to the development of Datalog as a widely accepted starting point for the development of practical DRDBs. The following strategies have been adopted in the design of DOOD systems:

Language Extension this approach consists of extending an existing deductive language model with object-oriented features. This approach is familiar from work on DRDBs such as LDL [NT89] or CORAL [RSS92], in which the syntax and semantics of Datalog were extended incrementally with negation, set terms, built in predicates, etc. In the language extension approach to DOODs, Datalog is generally an ancestor of the DOOD language, which is extended to support identity, inheritance, etc. Note that this strategy does not imply the existence of an earlier deductive database implementation or syntactic conformity with existing languages.

Language Integration here, a deductive language is integrated with an imperative programming language in the context of an object model or type system. In this strategy, the resulting system supports a range of standard object- oriented mechanisms for structuring both data and programs, while allowing different and complementary programming paradigms to be used for different tasks, or for different parts of the same task. The idea of integrating deductive and imperative language constructions for different parts of a task was pioneered in the Glue-Nail DRDB [DMP93], and later adapted for object-oriented databases.

Language Reconstruction an object model is reconstructed following the rationale of Reiter [Rei84], creating a new logic language that includes object- oriented features. In this strategy, the goal is to develop an object logic that captures the essentials of the object-oriented paradigm and that can also be used as a deductive programming language in DOODs. This is a revolutionary approach in the sense that much of the associated implementation technology may have to be conceived from scratch. The driving force behind the language reconstruction strategy is an argument that language extensions fail to combine object-orientation and logic successfully [KLW95], by losing declarativeness through the introduction of extra-logical features, or by failing to capture all aspects of the object-oriented model. The major drawbacks of reconstruction are the difficulty of providing efficient implementations of all the features of the object logic and the lack of agreement on the target model to be formalised.

4.3 Remarks & Outlook

All the models and technologies presented in this chapter influences the LVLP, in particular in how the model is conceived and designed. The LVLP work builds upon the general notion of *label* as defined by Gabbay [Gab96], and adopts the techniques introduced by Holzbaur [Hol92] to develop a generalisation of LP where labels are exploited to define computations in domain-specific contexts. Our characterisation can be viewed as a generalisation of the aforementioned approaches, blending the benefits of labels and LP so as to enable the very intrinsic nature of distributed situated intelligence. Indeed, LVLP allows heterogeneous devices in the IoT to have specific application goals and manage specific sorts of information, enabling reactivity to environment change while capturing diverse logic and domains.

Among the many differences w.r.t. the approaches described in this chapter is the fact that the approach proposed in this thesis does not change the basic of the logic language, which remains the same, but allows for different specific extensions tailored to local needs.

Chapter 5

Labelled Variables in Logic Programming

In order to enable logic programming to deal with the diversity of pervasive systems, where many heterogeneous, domain-specific computational models could benefit from the power of symbolic computation, we explore the expressive power of labelled systems. To this end, we define a new notion of truth for logic programs extended with *labelled variables* interpreted in non-Herbrand domains—where, however, terms maintain their usual Herbrand interpretations.

First, a model for labelled variables in logic programming is defined. Then, the fixpoint and the operational semantics are presented and their equivalence is formally proved. A meta-interpreter implementing the operational semantics is also introduced, followed by some case studies aimed at showing the effectiveness of our approach in selected scenarios. Accordingly, in the following we present the LVLP vision (Section 5.1). Then, we introduce the LVLP framework, moving from the definition of the theoretical model (Section 5.2) to the fixpoint and operational semantics (Section 5.3): we discuss correctness, completeness, and their equivalence is formally proved. Then, we present the technology (Section 5.4) implemented on the top of tuProlog system where a meta-interpreter implementing the operational semantics is discussed, which is exploited to illustrate LVLP through some case studies in different domains (Subsection 5.4.2).

5.1 Vision

The requirement for intelligence in pervasive systems is ubiquitous: computation surrounds us, devices and software components are required to behave intelligently, by understanding their own goals as well as the context where they work; integration of software components is supposed to add further (social) intelligence, possibly through coordination [Cas98]. This is the case, for instance, of the IoT[GBMP13, AIM10, FGRS14], where physical objects are networked, and are required to understand each other, learn, understand situations, and understand us [LMMZ17]—in short, our everyday objects are expected to be(come) intelligent in the Internet of Intelligent Things (IoIT) [ASF+14].

In the overall, IoIT scenarios mandate for distributed and situated *micro-intelligence*, where huge numbers of small units of computation, situated within a spatially-distributed environment, are required to behave in a smart way, and need to cooperate in order to achieve a coherent and intelligent social behaviour. However, engineering effective *distributed situated intelligence* is far from trivial, mostly due to *(i)* the huge amount of data, information, and knowledge to handle, *(ii)* the need for adaptation and self-management, *(iii)* the requirements of resource constrained devices, *(iv)* the heterogeneity of models and technologies against interoperability, and *(v)* the many diverse specific domains to be integrated.

Along that line, the goal of the LVLP model is to exploit the potential of LP and its extensions as sources of micro-intelligence for IoIT scenarios, in particular to deal with the domain-specific aspects. The LVLP domain-specific perspective further emphasises the role of situatedness, already brought along by spatial distribution of components in pervasive systems.

Accordingly, lightweight and interoperable LVLP Prolog engines could be distributed even on resource-constrained devices [DOC13]: multiple logic theories would then be scattered around, encapsulated in each engine, and associated to individual computational devices and things in the IoT. As a result, each logic theory is conceived as *situated*, and represents what is *locally* true, according to a simple paraconsistent overall interpretation. The LP resolution process is then local to each theory / engine, so it is both standard and consistent [Rob65]. Thus, LVLP allows in principle logic-based micro-intelligence to be encapsulated within devices of any sort, and make them work together in groups, aggregates, and societies, by promoting features such as observability, malleability, understandability, and formalisability via LP.

5.2 Model

Let $\mathscr{C}$ be the set of *constants*, with $c_1, c_2 \in \mathscr{C}$ being two generic constants. Let $\mathscr{V}$ be the set of *variables*, with $v_1, v_2 \in \mathscr{V}$ being two generic variables. Let $\mathscr{F}$ be the set of *function symbols*, with $f_1, f_2 \in \mathscr{F}$ being two generic function symbols; each $f \in \mathscr{F}$ is associated with arity $ar(f) > 0$, stating the number of function arguments. Let $\mathscr{T}$ be the set of *terms*, with $t_1, t_2 \in \mathscr{T}$ being two generic terms. Terms can be either simple – a constant (e.g., c_1) and a variable (e.g., v_2) are both *simple terms* – or compound—a functor of arity n applied to n terms (e.g., $f_1(c_2, v_1)$) is a *compound term*. A term is said *ground* if it does not contain variables. Let $\mathscr{H}$ denotes the set of ground terms, also known as the *Herbrand universe*.

A *model* for *Labelled Variables in Logic Programming* (LVLP) is defined as a triple $\langle \mathscr{B}, f_L, f_C \rangle$, where

- $\mathscr{B} = \{\beta_1, \ldots, \beta_n\}$ is the set of *basic labels*, the basic entities of the *domain of labels*
- $\mathscr{L} \subseteq \wp(\mathscr{B})$ is the set of *labels*, where each label $\ell \in \mathscr{L}$ is a subset of $\mathscr{B}$
- $f_L : \mathscr{L} \times \mathscr{L} \longrightarrow \mathscr{L}$ is the *(label-)combining function* computing a new label from two given ones
- $f_C : \mathscr{H} \times \mathscr{L} \longrightarrow \{\mathsf{true}, \mathsf{false}\}$ is the *compatibility function*, assessing the compatibility between a ground term and a label when interpreted in the domain of labels
- a *labelled variable* is a pair $\langle v, \ell \rangle$ associating label $\ell \in \mathscr{L}$ to variable $v \in \mathscr{V}$
- a *labelling* is a set of *labelled variables*

$\mathscr{B}$ can be either finite or infinite—in the latter case, with the two extra requirements that *(i)* each label ℓ can be represented finitely, including the new labels generated by the combining function f_L, and *(ii)* the compatibility function f_C can argue over the representation. Also, for the sake of simplicity, a "singleton" label $\{\beta\}$ where $\beta \in \mathscr{B}$ will be written just as β henceforth, and a "singleton" labelling $\{\langle v, \ell \rangle\}$ will be written as $\langle v, \ell \rangle$, and as v^ℓ in the examples.

Finally, we require that f_L is associative, commutative, and idempotent, with the empty set as its neutral element—namely:

$$f_L(\ell_1, f_L(\ell_2, \ell_3)) = f_L(f_L(\ell_1, \ell_2), \ell_3), f_L(\ell_1, \ell_2) = f_L(\ell_2, \ell_1), f_L(\ell, \ell) = \ell$$

Accordingly, in order to simplify notation, in the following we will simply write $f_L(\ell_1, \ldots, \ell_{n-1}, \ell_n)$ instead of $f_L(\ell_1, f_L(\ldots, f_L(\ell_{n-1}, \ell_n) \ldots))$.

Details on f_C and f_L are provided in the remainder of the Chapter, in particular in Subsection 5.2.2.

5.2.1 Programs, clauses, unification

An LVLP program is a collection of LVLP *rules* of the form

$$\mathit{Head} \leftarrow \mathit{Labelling}, \mathit{Body}.$$

to be read as "*Head* if *Body* given *Labelling*". There, *Head* is an atomic formula, *Labelling* is the list of labelled variables in the clause, and *Body* is a list of atomic formulas.

As in standard LP [Kow83, Bra13], an atomic formula (or atom) has the form $p(t_1, \ldots, t_m)$, where p is a predicate symbol and t_i are terms. Atom $p(t_1, \ldots, t_m)$ is said *ground* if $t_1, \ldots, t_n$ are ground. Predicate symbols represent relations defined by a logic program, whereas terms represent the elements of the domain. $\mathscr{H}_B$ is the *Herbrand base*, namely the set of all ground atoms of whose argument terms are in $\mathscr{H}$. Every variable occurring in a clause is universally quantified, and its scope is the clause in which the variable occurs.

An essential LP mechanism is represented by *unification*, involving two different terms that are supposed to refer to the same domain element. While discussing LP unification is out of the scope of this work (we refer the reader to [Kow83, Bra13] for the basics of LP), any extension to LP needs to define its own unification rules.

Thus, Table 5.1 reports the unification rules for LVLP. Since, by design, only variables can be labelled, the only case to be added to the standard unification table is represented by labelled variables. There, given two generic LVLP terms, the unification result is represented by the extended tuple

$$(\mathsf{true}/\mathsf{false}, \theta, \ell)$$

where true/false represents the existence of an answer, θ is the *most general unifier* (*mgu*), and ℓ is the new label associated to the unified variables defined by the (label-)combining function f_L. In order to lighten the notation, undefined elements in the tuple (i.e., labels or substitutions that make no sense in a given case) are omitted in Table 5.1.

Taking into account all the variables of a goal, a solution for a LVLP computation is represented by the extended tuple

$$(\mathsf{true}/\mathsf{false}, \Theta, A)$$

where Θ represents the *mgu* for all the variables, and A represents the corresponding labelling.

5.2.2 Compatibility

Expressing the solution of the labelled variables program as a tuple (true/false, Θ, A) implicitly assumes that the LP computation, whose answer is given by Θ, and the label computation, whose answer is given by A, can be read somehow independently from each other. So, whereas any computed label-variable association could be acceptable as far as LP is concerned (where symbols are uninterpreted), some label-variable association could be actually unacceptable when interpreted in the domain of labels.

To formalise such a notion of acceptability, the *compatibility function* f_C is defined as follows:

$$f_C : \mathscr{H} \times \mathscr{L} \longrightarrow \{\mathsf{true}, \mathsf{false}\}$$

In particular, given a a ground term $t \in \mathscr{H}$ and label $\ell \in \mathscr{L}$:

$$f_C(t, \ell) = \begin{cases} \mathsf{true} & \text{if there exists at least one element of the domain of labels which the interpretations of } t \text{ and } \ell \text{ both refer to} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Example 1 illustrates an application scenario where variables are labelled with their admissible numeric interval, formalising the f_L and f_C functions accordingly.

Example 1. LVLP with numeric intervals

As a simple LVLP example, let us suppose that logic variables span over integer domains, and are labelled with their admissible numeric intervals. The combining function f_L, which embeds the scenario-specific label semantics, is supposed to intersects intervals—that is, given two labels $\ell_1 = \{\beta_{11}, \dots, \beta_{1n}\}$ and $\ell_2 = \{\beta_{21}, \dots, \beta_{2m}\}$, the resulting label ℓ_3 is the intersection of ℓ_1 and ℓ_2:

$$\ell_3 = f_L(\ell_1, \ell_2) = f_L(\{\beta_{11}, \dots \beta_{1n}\}, \{\beta_{21}, \dots \beta_{2m}\}) =$$
$$\{(\beta_{11} \cap \beta_{21}), \cdots, (\beta_{11} \cap \beta_{2m}), \dots, (\beta_{1n} \cap \beta_{21}), \cdots, (\beta_{1n} \cap \beta_{2m})\}$$

Here the LP computation aims at computing numeric values, while the label computation aims at computing admissible numeric intervals for logic variables.

In principle, since the LP computation and the label computation proceed independently, the solution tuple ($\mathsf{true/false}$, Θ, A) could also describe situations such as $(\mathsf{true}, X/3, \langle X, [4,5]\rangle)$, where logic variable X would be associated to both value 3 and label $[4,5]$. However, if the numeric intervals are to be interpreted as the boundaries for acceptable values of LP variables, such labelling would be inconsistent, and the system should reject such a solution as *incompatible*.

This is what the compatibility function f_C is for: $f_C(t, \ell)$ connects the LP and the label universes by checking whether ground term $t \in \mathscr{H}$ is *compatible* with label $\ell \in \mathscr{L}$. In particular, in our example $f_C(t, \ell)$ is supposed to be true only if t belongs to the interval represented by ℓ: in this case, $f_C(3, [4,5])$ should be reasonably return false, thus rejecting labelling $\langle X, [4,5]\rangle$, with $X = 3$, as incompatible.

Summing up, the result of a LVLP program can be written as

$$((\mathsf{true/false}) \wedge f_C(\Theta, A), \Theta, A)$$

meaning that the truth value potentially computed by the LP computation can be restricted – i.e., forced to false – by $f_C(\Theta, A)$; in turn, this is just a convenient shortcut for the conjunction of all $f_C(t, \ell)$ $\forall (t, \ell)$ pairs, where ℓ and t are such that $\langle v, \ell\rangle \in A$ and $v/t \in \Theta$. Of course, in case of independent domains, $f_C(t, \ell)$ is merely true $\forall t$ and $\forall \ell$.

	constant c_2	*variable* v_2	*labelled variable* $\langle v_2, \ell_2 \rangle$	*compound term* s_2
constant c_1	if $c_1 = c_2$ then true else false	true, $\{v_2/c_1\}$	if $f_C(c_1, \ell_2) =$ true then true, $\{v_2/c_1\}$, ℓ_2 else false	false
variable v_1	true, $\{v_1/c_2\}$	true, $\{v_1/v_2\}$	true, $\{v_1/v_2\}, \ell_2$	if v_1 does not occur in s_2 then true, $\{v_1/s_2\}$ else false
labelled variable $\langle v_1, \ell_1 \rangle$	if $f_C(c_2, \ell_1) =$ true then true, $\{v_1/c_2\}$, ℓ_1 else false	true, $\{v_1/v_2\}, \ell_1$	if $f_L(\ell_1, \ell_2) \neq \emptyset$ then true, $\{v_1/v_2\}$, $f_L(\ell_1, \ell_2)$ else false	if v_1 does not occur in s_2, and $f_L(\ell_1, \ell'_1, ..., \ell'_n) \neq \emptyset$ where $\ell'_1, ...\ell'_n$ are the labels in s_2 then true, $\{v_1/s_2\}$, $f_L(\ell_1, \ell'_1, ..., \ell'_n)$ else false
compound term s_1	false	if v_2 does not occur in s_1 then true, $\{v_2/s_1\}$ else false	if v_2 does not occur in s_1, and $f_L(\ell_2, \ell'_1, ..., \ell'_n) \neq \emptyset$ where $\ell'_1, ...\ell'_n$ are the labels in s_1 then true, $\{v_2/s_1\}, f_L(\ell_2, \ell'_1, ..., \ell'_n)$, else false	if s_1, s_2 have same functor / arity, and their arguments (recursively) unify then true else false

Table 5.1: *Unification rules in LVLP, adopting standard LP unification rules and representation*

5.3 Foundations & Semantics

In order to maintain the basic theoretical results of LP, such as the equivalence of denotational and operational semantics, labels domains must support tests and operations on labels.

To this end, Subsection 5.3.1 defines the denotational (fixpoint) semantics in the context of labels domain (under reasonable requirements for f_L), while Subsection 5.3.2 discusses the operational semantics of the model through an abstract state machine.

5.3.1 Fixpoint semantics

Let us call $\mathcal{X} = (\mathscr{H}, \mathscr{L})$ a *LVLP domain*, and define the notion of $\mathcal{X}$-interpretation I as a set of pairs of the form

$$\big\langle p(t_1, \ldots, t_n), [\ell_1, \ldots, \ell_n] \big\rangle$$

where $p(t_1, \ldots, t_n)$ is a ground atom, and $\ell_1, \ldots, \ell_n$ are labels s.t. for $i = 1, \ldots, n$ the term t_i is *compatible* with the label ℓ_i, i.e., $f_C(t_i, \ell_i) = \mathsf{true}$. Truthness of f_C is based on the LVLP domain $\mathcal{X}$. With a slight abuse of notation, we write $\mathcal{X} \models [\langle t_1, \ell_1 \rangle, \ldots, \langle t_n, \ell_n \rangle]$ iff $\bigwedge_{i=1}^{n} f_C(t_i, \ell_i) = \mathsf{true}$. We also write $\mathcal{X} \models \langle p(t_1, \ldots, t_n), [\ell_1, \ldots, \ell_n] \rangle$ if p is a predicate symbol and $\bigwedge_{i=1}^{n} f_C(t_i, \ell_i) = \mathsf{true}$.

We denote as Λ the part of clause body that stores labelling. Without loss of generality we assume that there is exactly one labelling for each variable in the head. We define the function $\widetilde{f_L}$ that extends f_L and takes as arguments, orderly, a rule

$$r = h \leftarrow \Lambda, b_1, \ldots, b_n$$

a labelling, and n lists of labels. The rule r is used here to identify multiple occurrences of the variables. Let us assume the variables in h are $x_1, \ldots, x_m$, and consider one of them, say x_i. If x_i occurs in h (and hence in Λ) and in (some of) $b_1, \ldots, b_n$ then the corresponding labels $\ell, \ell_{1,i}, \ldots, \ell_{n,i}$ for x_i are retrieved from Λ (if x_i does not occur in b_j simply we do not consider such contribution). Then $\ell'_i = f_L(\ell, f_L(\ell_{1,i}, \ldots, f_L(\ell_{n-1,i}, \ell_{n,i})))$ is computed and the pair $\langle x_i, \ell' \rangle$ is returned. This is done for all variables $x_1, \ldots, x_m$ occurring in the head h and the list $[\ell'_1, \ldots, \ell'_m]$ is returned.

The denotational semantics is based on the one-step consequence functions T_P defined as:

$$T_P(I) = \left\{ \begin{array}{ll} \langle p(\widetilde{t}), \widetilde{\ell}\ \rangle : & \\ r = p(\widetilde{x}) \leftarrow \Lambda_{\widetilde{x}}\ ,\ b_1, \ldots, b_n & (1) \\ \text{where r is a fresh renaming of a rule of } P, & \\ v \text{ is a valuation on } \mathscr{H} \text{ such that } v(\widetilde{x}) = \widetilde{t}, & (2) \\ \bigwedge_{i=1}^{n} \exists\ \widetilde{\ell_i} \text{ s.t. } \langle v(b_i), \widetilde{\ell_i} \rangle \in I, & (3) \\ \mathcal{X} \models \Lambda_{\widetilde{t}} \wedge \bigwedge_{i=1}^{n} f_C\big(v(b_i), \widetilde{\ell_i}\big), & (4) \\ \widetilde{\ell} = \widetilde{f_L}\big(r, \Lambda_{\widetilde{t}}, \widetilde{\ell_1}, \cdots, \widetilde{\ell_n}\big) & (5) \end{array} \right\}$$

where $\Lambda_{\widetilde{t}} = v(\Lambda_{\widetilde{x}}) = [\langle t_1, \ell_1 \rangle, \ldots, \langle t_n, \ell_m \rangle]$ if $\Lambda_{\widetilde{x}} = [\langle x_1, \ell_1 \rangle, \ldots, \langle x_m, \ell_m \rangle]$. Notice that the condition $\mathcal{X} \models \bigwedge_{i=1}^{n} f_C\big(v(b_i), \widetilde{\ell_i}\big)$ in line 4 is always satisfied when T_P is used bottom-up, starting from $I = \emptyset$.

For the sake of convenience, unspecified labels are assumed to be read as the *any* label, defined as the neutral element of f_L: in this way, any standard (i.e. non labelled) LP variable can be read as implicitly labelled with such *any* label—represented as $\diamond$ henceforth.

Example 1 reports an example of computation of the least fixpoint of T_P in a simple case. There, and in the following examples, `=/2` represents the equality operator of LVLP, whose behaviour is described in Table 5.1 (unification rules), and can be summarised as:

$$\mathtt{X} = \mathtt{Y} : -[<\mathtt{X}, \diamond>, <\mathtt{Y}, \diamond>], \mathtt{X} =_{\mathtt{LP}} \mathtt{Y}$$

where $=_{\mathtt{LP}}$ is the `=/2` standard LP unification operator.

Example 1. Computing the least fixpoint of T_P in a simple case

Let us consider the LVLP program P:

```
    r(0). r(1). r(2). r(3). r(4). r(5). r(6). r(7). r(8). r(9).
    q(Y,Z) :- Y^[[2,4]], Z^[[3,8]], Y=Z, r(Y), r(Z).
    p(X,Y,Z) :- X^[[0,3]], Y^[◇], Z^[◇], X=Y, q(Y,Z).
```

where $\diamond$ is used as a shortcut for any basic label (any interval).
Let us consider the interpretation $I_0 = \emptyset$. Then, the next interpretation I_1 can be obtained as:

$$I_1 = T_P(I_0) = \big\{\langle r(0), [\diamond] \rangle, \ldots, \langle r(9)[\diamond] \rangle\big\}$$

Applying T_P to I_1 leads to I_2:

$$I_2 = T_P(I_1) = I_1 \cup \big\{\langle q(3,3), [[3,4],[3,4]] \rangle, \langle q(4,4), [[3,4],[3,4]] \rangle\big\}$$

One further step leads to I_3, which is also the least fixpoint of T_P:

$$I_3 = T_P(I_2) = I_2 \cup \big\{\langle p(3,3,3), [[3],[3],[3]] \rangle\big\}$$

The example above shows how LVLP on a domain $\mathcal{X}$ – $LVLP(\mathcal{X})$ – looks like $CLP(\mathcal{X})$:

in fact, $[\langle Y, [2,4]\rangle, \langle Z, [3,8]\rangle]$ can be interpreted as the constraints $Y \in [2,4], Z \in [3,8]$. However, this does not hold for all the label domains, since LVLP aims at covering domains beyond the reach of constraint logic programming.

This is the case, for instance, of Example 2, where labels are words in the WordNet lexical database [Fel06]. There, the combining function f_L is supposed to find and return a WordNet synset compatible with both the given labels, or fail otherwise: for instance, if ℓ_1 = ‘*pet*′ and ℓ_2 = ‘*domestic cat*’, the new label generated by f_L could be [‘*cat*’, ‘*domestic cat*’, ‘*pet*’, ‘*mammal*’]. The compatibility function f_C is always `true`, since any animal name is considered compatible with any animal group.

Example 2. T_P in the WordNet case

In this example, labels are words describing the object represented by the variable. The combination of two different labels (performed by the combining function f_L) returns a new label only if the two labels have a lexical relation, or fail otherwise. The decision is based on the WordNet network [Fel06], a large lexical database of English where nouns, verbs, adjectives, and adverbs are grouped into sets of cognitive synonyms (synsets). Synsets are interlinked by means of conceptual-semantic and lexical relations: the resulting network of meaningfully related words and concepts can be navigated. WordNet superficially resembles a thesaurus, in that it groups words together based on their meanings.
In this first showcase, some WordNet groups are collected and stored in the knowledge base a priori, but a dynamic consultation to WordNet could be implemented. Let program P be represented by the following facts—where `wordnet_fact` is a simulated wordnet synset, while `animal(Name)` is a predicate describing an animal's name:

```
wordnet_fact([‘dog’,‘domestic dog’,‘canis’,‘pet’,‘mammal’]).
wordnet_fact([‘cat’, ‘domestic cat’, ‘pet’, ‘mammal’]).
wordnet_fact([‘fish’,‘aquatic vertebrates’, ‘vertebrate’]).
wordnet_fact([‘frog’,‘toad’, ‘anuran’, ‘batrachian’]).

pet_name(‘minnie’).
fish_name(‘nemo’).
animal(X) :- X^[‘pet’], pet_name(X).
animal(X) :- X^[‘fish’], fish_name(X).
```

The combining function f_L is supposed to find and return a WordNet synset compatible with both labels. So, if $\ell_1 = pet$ and $\ell_2 = \text{‘}domestic\ cat'$, the new generated label ℓ_3 is $[\text{‘}cat', \text{‘}domestic\ cat', \text{‘}pet', \text{‘}mammal']$. The compatibility function f_C in this scenario is always `true`, since any animal name is considered compatible with any animal group.
In order to show the construction of T_P, let us consider the interpretation $I_0 = \emptyset$. Then, the subsequent step I_1 can be computed as:

$$I_1 = T_P(I_0) = \{ \quad \langle pet\_name(\text{‘}minnie'), [\diamond]\rangle, \langle fish\_name(\text{‘}nemo')[\diamond]\rangle\}$$

Now, let us apply T_P to I_1 to compute I_2:

$$\begin{aligned}&I_2 = T_P(I_1) = I_1 \cup \{\\ &\quad \langle animal(\text{‘}minnie'), [[\text{‘}dog', \text{‘}domestic\ dog', \text{‘}canis', \text{‘}pet', \text{‘}mammal']]\rangle,\\ &\quad \langle animal(\text{‘}minnie'), [[\text{‘}cat', \text{‘}domestic\ cat', \text{‘}pet', \text{‘}mammal']]\rangle,\\ &\quad \langle animal(\text{‘}nemo'), [[\text{‘}fish', \text{‘}aquatic\ vertebrates', \text{‘}vertebrate']]\rangle\}\end{aligned}$$

which is also the least fixpoint of T_P.

We prove that T_P, if applied bottom-up starting from the empty interpretation, always leads to a minimum fixpoint (Corollary 5.3.3). Such an interpretation is the denotational semantics of the program P. In order to achieve that result, we need to prove that T_P is monotonic and continuous, and use the Knaster-Tarksi and Kleene's fixpoint theorems (Proposition 5.3.2).

An interpretation, for being a model, should satisfy the meaning of every rule, namely: if for a given valuation of the variables the body is considered `true` by the interpretation, then the head must also be `true`. We state this property formally:

Definition 5.3.1 *An interpretation I is a model of a program P if for each rule $r = p(\widetilde{x}) \leftarrow \Lambda_{\widetilde{x}}\ ,\ b_1, \ldots, b_n$ of P and each valuation v of the variables in r on $\mathscr{H}$ (let us denote with $\widetilde{t} = v(\widetilde{x})$) it holds that if*

- *for $i = 1, \ldots, n$ there are $\langle v(b_i), \widetilde{\ell_i} \rangle \in I$*
- *such that $\mathcal{X} \models f_C(v(b_i), \widetilde{\ell_i})$ and*
- *$\mathcal{X} \models \Lambda_{\widetilde{t}}$*

then it holds that $\big\langle p(\widetilde{t}), \widetilde{f_L}\big(r, \Lambda_{\widetilde{t}}, \widetilde{\ell_1}, \cdots, \widetilde{\ell_n}\big)\big\rangle \in I$.

Proposition 5.3.2 *Given a LVLP program P and a LVLP domain $\mathcal{X}$, T_P is (1) monotonic and (2) (upward) continuos, and (3) $T_P(I) \subseteq I$ iff I is a model of P.*

Proof: (1) Let I and J be two interpretations such that $I \subseteq J$. We need to prove that $T_P(I) \subseteq T_P(J)$. If $a = \langle p(\widetilde{t}), \widetilde{\ell}\ \rangle \in T_P(I)$, there are a clause $r \in P$, a valuation v on $\mathscr{H}$ for the variables in r and n elements $\langle v(b_i), \widetilde{\ell_i} \rangle \in I$ satisfying the remaining conditions. Since $I \subseteq J$, they belong to J as well: so, $a \in T_P(J)$.

(2) Let us consider a chain of interpretations $I_0 \subseteq I_1 \subseteq \ldots$: we need to prove that $T_P\big(\bigcup_{k=0}^{\infty} I_k\big) = \bigcup_{k=0}^{\infty} T_P(I_k)$.
($\subseteq$) Let $a = \langle p(\widetilde{t}), \widetilde{\ell}\ \rangle \in T_P\big(\bigcup_{i=0}^{\infty} I_k\big)$. Thus, there are a clause $r \in P$, a valuation v on $\mathscr{H}$ for the variables in r, and n elements $\langle v(b_i), \widetilde{\ell_i} \rangle \in \bigcup_{k=0}^{\infty} I_k$ satisfying the remaining conditions. This means that there are $j_1, \ldots, j_n$ such that for $i = 1, \ldots, n$ $\langle v(b_i), \widetilde{\ell_i} \rangle \in I_{j_i}$. Now let $j = \max\{j_1, \ldots, j_n\}$: since $I_0 \subseteq I_1 \subseteq \cdots \subseteq I_j$, all $\langle v(b_i), \widetilde{\ell_i} \rangle \in I_j$. Thus $a \in T_P(I_{j+1})$ and henceforth $a \in \bigcup_{k=0}^{\infty} T_P(I_k)$.
($\supseteq$) Let $a = \langle p(\widetilde{t}), \widetilde{\ell}\ \rangle \in \bigcup_{k=0}^{\infty} T_P(I_k)$. This means that there is j such that $a \in T_P(I_j)$. Then, due to monotonicity, $a \in T_P\big(\bigcup_{k=0}^{\infty} I_k\big)$.

(3) Let $T_P(I) \subseteq I$ and let $r = p(\widetilde{x}) \leftarrow \Lambda_{\widetilde{x}}\ ,\ b_1, \ldots, b_n$ be a generic clause of P, and v be a generic valuation on $\mathscr{H}$ of all the variables of r. If we assume that $\langle v(b_i), \widetilde{\ell_i} \rangle \in I$ for

$i = 1, \ldots, n$, and that $\mathcal{X} \models f_C(v(b_i), \widetilde{\ell}_i)$ and $\mathcal{X} \models \Lambda_{\widetilde{t}}$, then the pair $h = v(p(\widetilde{x}), \widetilde{\ell}) \in T_P(I)$ by definition of T_P—and $\widetilde{\ell}$ is the same of the definition of model. Since $T_P(I) \subseteq I$ then $h \in I$, therefore (since r and v are chosen in general) I is a model of P.

On the other hand, if I is a model of P we prove that $T_P(I) \subseteq I$. Let $a = \langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P(I)$. This means that there is a rule $r = p(\widetilde{x}) \leftarrow \Lambda_{\widetilde{x}}$, $b_1, \ldots, b_n$ of P and a valuation v of the variables in r on $\mathscr{H}$ such that for $i = 1, \ldots, n$ there are $\big(v(b_i), \widetilde{\ell}_i\big) \in I$ and $\mathcal{X} \models \Lambda_{\widetilde{t}}$ and $\widetilde{\ell} = \widetilde{f}_L\big(r, \Lambda_{\widetilde{t}}, \widetilde{\ell}_1, \cdots, \widetilde{\ell}_n\big)\big)$, such that $\mathcal{X} \models f_C(v(b_i), \widetilde{\ell}_i)$. Since I is a model and $\mathcal{X} \models f_C(v(b_i), \widetilde{\ell}_i)$ then $a \in I$. □

Let $T_P \uparrow \omega$ be defined as usual: $T_P \uparrow \omega = \bigcup\{T_P \uparrow k : k \geq 0\}$, where $T_P \uparrow 0 = \emptyset$ and $T_P \uparrow (n+1) = T_P(T_P \uparrow n)$. Then

Corollary 5.3.3 *T_P has a least fixpoint.*

Proof: Being T_P monotonic, the Knaster-Tarksi theorem ensures that it admits a least (and a greatest) fixpoint. Being (upward) continuous, Kleene's fixpoint Theorem states that $T_P \uparrow \omega$ is the least fixpoint. □

5.3.2 Operational semantics

In this section we define the operational interpretation of labels. Our approach is inspired by the methodology introduced for constraint logic programming (CLP) [Coh90, ICW93, JM94]: accordingly, we define the LVLP abstract state machine based on that suggested by Colmerauer for Prolog III [Col90]. We define a labelled-machine state σ as the triplet:

$$\sigma = \langle t_0\ t_1 ... t_n, W, \Lambda \rangle$$

in which $t_0\ t_1 ... t_n$ is the list of terms (goals), W is the current list of variable bindings, Λ is the current labelling on W.

An inference step for the machine consists of making a transition from the state σ to a state σ' by applying a program rule. If $m \geq 0$ and Λ' is a set of labelled variables, a program rule

$$s_0 \leftarrow \Lambda', s_1, s_2, \ldots, s_m$$

is applicable if the following conditions hold:

- $\exists\, mgu\, \theta$ such that $\theta(t_0) = \theta(s_0)$, and
- $\widetilde{f}_{\theta L}(\Lambda, \theta(\Lambda')) \neq \emptyset$

Function $\widetilde{f}_{\theta L}$ is a generalisation of f_L taking as arguments two labellings. If x_i occurs in both Λ and $\theta(\Lambda')$ with labels ℓ and ℓ', $\ell'' = f_L(\ell, \ell')$ is first calculated, then the labelled variable $\langle x_i, \ell'' \rangle$ is returned, provided that $f_C(\theta(x_i), \ell'') = \mathsf{true}$; otherwise, false is returned. Thus the new state becomes:

$$\sigma' = \langle \theta(s_1, \ldots, s_m, t_1, \ldots, t_n), W' = W \circ \theta, \Lambda'' = \widetilde{f}_{\theta L}(\Lambda, \theta(\Lambda')) \rangle$$

where $\circ$ applies the classical composition of substitutions.

A solution is found when a final state is reached. The final state has the form:

$$\sigma_f = (\epsilon, W_f, \Lambda_f)$$

where ϵ is the empty sequence, W_f is the final list of variables and bindings, and Λ_f is the corresponding labelling. A sequence of applications of inference steps is said to be a *derivation*. A derivation is *successful* if it ends in a final state, or *failing* if it ends in a non-final state where no further inference step is possible.

Proposition 5.3.4 *Let $p(\widetilde{x})$ be an atom, v a valuation on $\mathscr{H}$ such that $v(\widetilde{x}) = \widetilde{t}$ where $\widetilde{t}$ are ground terms, and $\widetilde{\ell}$ a list of labels. Then there is a successful derivation for $\langle p(\widetilde{t}), v, \langle v(\widetilde{x}), \widetilde{\ell} \rangle \rangle$ iff $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow \omega$.*

Proof: *In the following we omit some standard details for the sake of brevity, please refer to, e.g., [JM94]*

$(\rightarrow)$. We prove the proposition by induction on the length k of the derivation. If $k = 0$ the result holds trivially.

For the inductive case, let us suppose that there is a successful derivation for $\langle p(\widetilde{t}), v, \langle v(\widetilde{x}), \widetilde{\ell} \rangle \rangle$ of $k + 1$ steps. Let us focus on the first step: there is a rule r: $s_0 \leftarrow \Lambda', s_1, s_2, \ldots, s_m$ such that $\theta(p(\widetilde{t})) = \theta(s_0)$ leading to the new state $\sigma = \langle \theta(s_1, s_2, \ldots, s_m), v \circ \theta, \Lambda'' = \widetilde{f}_{\theta L}(\Lambda, \theta(\Lambda')) \rangle$, where $f_C(\Lambda'') = \mathsf{true}$, that admits a successful derivation of k steps.

Consider now the states $\sigma_1, \ldots, \sigma_m$ defined as $\sigma_i = \langle \theta(s_i), v \circ \theta, \Lambda''_i \rangle$ where Λ''_i is the restriction of Λ'' to the variables in s_i. Since σ admits a successful derivation of $k + 1$ steps, each σ_i should admit a successful derivation of at most k steps.

If for all $i \in \{1, \ldots, m\}$, $\theta(s_i)$ is ground, then, by inductive hypothesis we have that $\langle \theta(s_i), \ell_i \rangle \in T_P \uparrow \omega$ where $\ell_i = \pi_2(\Lambda''_i)$, and hence that there are h_is such that $\langle \theta(s_i), \ell_i \rangle \in T_P \uparrow h_i$. Since T_p is monotonic, all of them belong to $T_P \uparrow h$ where $h = \max_{i=1,\ldots,m} h_i$. Then, by applying T_P considering the rule r, since we already know that $\Lambda'' = \widetilde{f}_{\theta L}(\Lambda, \theta(\Lambda'))$, and $f_C(\Lambda'') = \mathsf{true}$, we have that $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow h + 1$, hence to $T_P \uparrow \omega$.

If for some i, $\theta(s_i)$ is not ground, the above property holds for any ground instantiation of the remaining variables and again the results follows.

$(\leftarrow)$. Now, let us analyse the converse direction. If $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow \omega$ this means that there is a $k \geq 0$ such that $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow k$.

Let us prove by induction on k. Again, if $k = 0$ the result holds trivially. Let us suppose now that $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow k + 1$. This means (by definition of T_P) that there is a rule r: $s_0 \leftarrow \Lambda', s_1, s_2, \ldots, s_m$ such that $s_0 = p(\widetilde{x})$ and there is a valuation u on $\mathscr{H}$ such that $u(s_0) = p(\widetilde{t})$ and that, in particular, $\langle s_1, \ell_1\rangle, \ldots, \langle s_m, \ell_m\rangle \in T_P \uparrow k$ (and f_L can be computed and f_C is true on these arguments). By inductive hypothesis, for $i \in \{1, \ldots, m\}$ there is a derivation, say, of h_i steps for $\sigma_i = \langle u(s_i), v \circ u, \langle u(s_i), \ell_i\rangle\rangle$

Since f_C is true on such arguments and f_L can be computed, the same holds for T_P: so, we have a derivation of $\sum_{i=1}^{m} h_i + 1$ steps for $\langle p(\widetilde{t}), v \circ u \circ \theta, \langle \widetilde{t}, \widetilde{\ell}\rangle\rangle$.

This completes the proof of the inductive step. □

5.4 Technology

5.4.1 Meta-interpreter

The operational semantics of LVLP is captured in the meta-interpreter shown in Listing 5.1: the code is developed in tuProlog [DOR05], a light-weight Prolog system whose (Java-based) design inherently enables the injection of Prolog programs within pervasive systems, as well as their integration with diverse programming languages and paradigms, over computing platforms of any sort [DOC13].

```
%%%% solve(+Goals, +LVarsIn, -LVarsOut)
%% Goals is the list of goals to solve
%% LVarsIn is the labelling on goals variables
%% LVarsOut is the final labelling on output variables
% termination condition
solve([], LVars, LVars) :- !. % for efficiency
% goal iterator
solve([Goal|Goals], LVarsIn, LVarsOut):- !,
  solve(Goal, LVarsIn, LVarsTempOut),
  solve(Goals, LVarsTempOut, LVarsOut).
% solve core
solve(Goal, LVarsIn, LVarsOut):-
  clause(Goal, LVars, Body),
  mergeLabels(LVarsIn, LVars, LVarsTempOut),
  solve(Body, LVarsTempOut, LVarsOut).
```

Listing 5.1: The LVLP meta-interpreter: the `solve/3` predicate.

The `solve/3` predicate[1] has three arguments (Listing 5.1), namely:

- the list of the goals to be processed
- the current labelling
- the new labelling updated by the goal resolution process

The `solve/3` predicate recursively calls itself to process the goal list, and exploits `clause/3` and `mergeLabels/3` to, respectively, handle single goals and combine labels: in particular,

[1]`solve/3` refers to standard meta-interpreter Prolog [SSW86]

`clause/3` finds a clause in the database whose head matches with `Goal` and returns both the `Body` of the clause and the labelling in the selected clause, `LVars`.

The core of the meta-interpreter is embedded in the `mergeLabels/3` predicate (detailed in Listing 5.2), which combines two sets of labels – the previous labelling, `LVarsIn`, and the labelling introduced by the current clause, `LVars` – into the new `LVarsTempOut`, or fails if no solution can be found. The generation of the new label is performed via the `label_generate/3` predicate, which embeds the combining function f_L, and is provided by the user according to the domain-specific features of the application scenario.

```
%%%% mergeLabels(+LVars1, +LVars2, -LVars3)
%% LVars1, LVars2, LVars3 are lists of labelled variables
%% LVars3 is obtained merging the labelled variables in LVars1 and LVars2
%% LVars1 and LVars2 are sorted according to the same criterion--e.g.,
%% alphabetically.
%% For all the variables that appear both in LVars1 and LVars2, the resulting
%% label is obtained using the combining function embedded in label_generate/3
% termination conditions
mergeLabels(LVars, [], LVars) :- !.
mergeLabels([], LVars, LVars) :- !.
% variable Var1 propagation in LVars3 if it is contained only in LVars1
mergeLabels([Var1^L1|LVars1], LVars2, [Var1^L1|LVars3]):-
  not_in(Var1,LVars2), !,
  mergeLabels(LVars1, LVars2, LVars3).
% variable Var2 propagation in LVars3 if it is contained only in LVars2
mergeLabels(LVars1, [Var2^L2|LVars2], [Var2^L2|LVars3]):-
  not_in(Var2,LVars1), !,
  mergeLabels(LVars1, LVars2, LVars3).
% generation of a new label if variable Var is contained both in LVars1 and
% LVars2
mergeLabels([Var^L1|LVars1], [Var^L2|LVars2], [Var^L3|LVars3]):-
  label_generate(L1, L2, L3),
  mergeLabels(LVars1, LVars2, LVars3).
% utility func not_in(+Var, +LVars) checks if the list LVars does not contain Var
not_in(_,[]).
not_in(X,[Y^_|_]) :- !, fail.
not_in(X,[_|T]) :- not_in(X,T).
```

Listing 5.2: The LVLP meta-interpreter: the `mergeLabels/3` predicate.

5.4.2 Case studies

In the following we show some LVLP computations based on our prototype rooted in Labelled `tuProlog` [CDO15], which exploits the meta-interpreter presented in Subsection 5.4.1—available on Bitbucket [tup01].

WordNet network

This example extends and implements the case study of Example 2. A label is a network of related words describing the semantic net of the object represented by the associated variable according to the WordNet lexical database [Fel06]. Listing in Figure 5.1 (top) shows the tuProlog implementation of the `label_generate/3` predicate embedding the combining function f_L. Here, the `label_generate` checks if ℓ_1 and ℓ_2 are contained in a common `wordnet_fact`.

Following our prototype syntax, `X^`*`label`* is a labelled variable denoting a logic variable `X` labelled with *`label`*—where *`label`* is a term in the set of admissible labels defined by the user. In a LVLP clause, the list of the labelled variables precedes the remaining part of the body. So, given the following LVLP program:

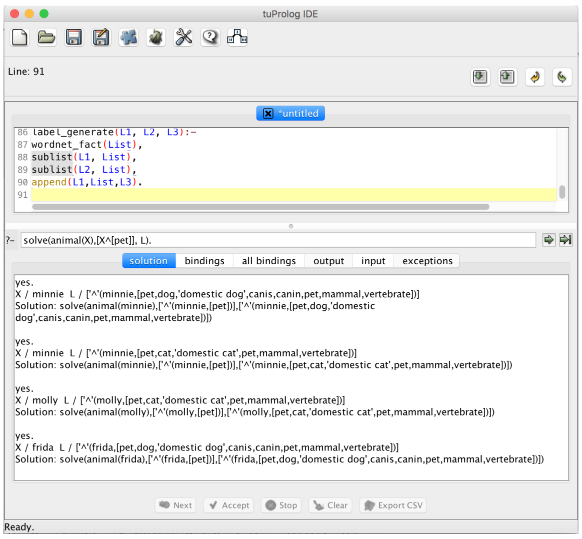
```
  wordnet_fact([‘dog’,‘domestic dog’,‘canis’,‘pet’,‘mammal’,‘vertebrate’]).
  wordnet_fact([‘cat’, ‘domestic cat’, ‘pet’, ‘mammal’, ‘vertebrate’]).
  wordnet_fact([‘fish’, ‘aquatic vertebrates’, ‘vertebrate’]).
  wordnet_fact([‘frog’, ‘toad’, ‘anuran’, ‘batrachian’]).
  animal(X) :- X^[‘pet’], X = ‘minnie’.
  animal(X) :- X^[‘fish’], X = ‘nemo’.
  animal(X) :- X^[‘cat’], X = ‘molly’.
  animal(X) :- X^[‘dog’], X = ‘frida’.
  animal(X) :- X^[‘frog’], X = ‘cra’.
```

the following query, looking for a `pet` animal, generates four solutions:

```
?- X^[‘pet’], animal(X).

yes. X / ‘minnie’
X^[‘dog’, ‘domestic dog’, ‘canis’, ‘pet’, ‘mammal’, ‘vertebrate’];
```

```
  %% label_generate(+L1, +L2, -L3) embedding f_L behaviour for WordNet groups
  label_generate(L1, L2, List):-
    wordnet_fact(List), sublist(L1, List), sublist(L2, List).
```



```
label_generate(L1, L2, L3):-
wordnet_fact(List),
sublist(L1, List),
sublist(L2, List),
append(L1,List,L3).
```

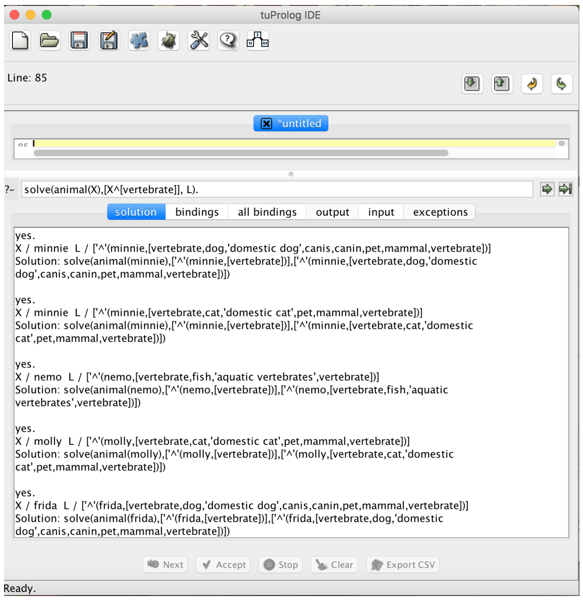
```
?- solve(animal(X),[X^[pet]], L).

yes.
X / minnie  L / ['^'(minnie,[pet,dog,'domestic dog',canis,canin,pet,mammal,vertebrate])]
Solution: solve(animal(minnie),['^'(minnie,[pet])],['^'(minnie,[pet,dog,'domestic
dog',canis,canin,pet,mammal,vertebrate])])

yes.
X / minnie  L / ['^'(minnie,[pet,cat,'domestic cat',pet,mammal,vertebrate])]
Solution: solve(animal(minnie),['^'(minnie,[pet])],['^'(minnie,[pet,cat,'domestic cat',pet,mammal,vertebrate])])

yes.
X / molly  L / ['^'(molly,[pet,cat,'domestic cat',pet,mammal,vertebrate])]
Solution: solve(animal(molly),['^'(molly,[pet])],['^'(molly,[pet,cat,'domestic cat',pet,mammal,vertebrate])])

yes.
X / frida  L / ['^'(frida,[pet,dog,'domestic dog',canis,canin,pet,mammal,vertebrate])]
Solution: solve(animal(frida),['^'(frida,[pet])],['^'(frida,[pet,dog,'domestic dog',canis,canin,pet,mammal,vertebrate])])
```



```
?- solve(animal(X),[X^[vertebrate]], L).

yes.
X / minnie  L / ['^'(minnie,[vertebrate,dog,'domestic dog',canis,canin,pet,mammal,vertebrate])]
Solution: solve(animal(minnie),['^'(minnie,[vertebrate])],['^'(minnie,[vertebrate,dog,'domestic
dog',canis,canin,pet,mammal,vertebrate])])

yes.
X / minnie  L / ['^'(minnie,[vertebrate,cat,'domestic cat',pet,mammal,vertebrate])]
Solution: solve(animal(minnie),['^'(minnie,[vertebrate])],['^'(minnie,[vertebrate,cat,'domestic
cat',pet,mammal,vertebrate])])

yes.
X / nemo  L / ['^'(nemo,[vertebrate,fish,'aquatic vertebrates',vertebrate])]
Solution: solve(animal(nemo),['^'(nemo,[vertebrate])],['^'(nemo,[vertebrate,fish,'aquatic
vertebrates',vertebrate])])

yes.
X / molly  L / ['^'(molly,[vertebrate,cat,'domestic cat',pet,mammal,vertebrate])]
Solution: solve(animal(molly),['^'(molly,[vertebrate])],['^'(molly,[vertebrate,cat,'domestic
cat',pet,mammal,vertebrate])])

yes.
X / frida  L / ['^'(frida,[vertebrate,dog,'domestic dog',canis,canin,pet,mammal,vertebrate])]
Solution: solve(animal(frida),['^'(frida,[vertebrate])],['^'(frida,[vertebrate,dog,'domestic
dog',canis,canin,pet,mammal,vertebrate])])
```

Figure 5.1: `label_generate/3` *example: WordNet case study*

```
yes. X / ‘minnie’
X^[‘cat’, ‘domestic cat’, ‘pet’, ‘mammal’, ‘vertebrate’];

yes. X / ‘molly’
X^[‘cat’, ‘domestic cat’, ‘pet’, ‘mammal’, ‘vertebrate’];

yes. X / ‘frida’
X^[‘dog’, ‘domestic dog’, ‘canis’, ‘pet’, ‘mammal’, ‘vertebrate’]
```

Looking instead for a less specific `vertebrate` produces five solutions:

```
?- X^[‘vertebrate’], animal(X).

yes. X = ‘minnie’
X^[‘dog’, ‘domestic dog’, ‘canis’, ‘pet’, ‘mammal’, ‘vertebrate’] ;

yes. X = ‘minnie’
X^[‘cat’, ‘domestic cat’, ‘pet’, ‘mammal’, ‘vertebrate’] ;

yes. X = ‘molly’
X^[‘cat’, ‘domestic cat’, ‘pet’, ‘mammal’, ‘vertebrate’] ;

yes. X = ‘frida’
X^[‘dog’, ‘domestic dog’, ‘canis’, ‘pet’, ‘mammal’, ‘vertebrate’] ;

yes. X = ‘nemo’
X^[‘fish’,‘aquatic vertebrates’, ‘vertebrate’]
```

A relevant aspect of LVLP is that labels are not subject to the single-assignment assumption: each time two labelled variables unify, their labels are processed and *combined* according to the user-defined function that embeds the desired computational model, and the resulting label is associated to the unified variable. Thus, while the LP model *per se* is left untouched, diverse computational models can be associated with, possibly influencing the result of a logic computation by *restricting* the set of admissible solutions according to each specific domain.

Dress selection

In the following example the application scenario is the selection from a wardrobe of a dress that is "similar enough" to a given colour. A fact `shirt(`*`Description`*`, `*`Colour`*`)` represents a shirt with of colour *`Colour`*, expressed as a triple of the form `rgb(`*`Red`*`,`*`Green`*`,`*`Blue`*`)` in *`Description`*.

For instance, shirts in a wardrobe could be represented as follows:

```
○ shirt(rgb(255,240,245), my_pink_blouse).
● shirt(rgb(255,222,173), old_yellow_tshirt).
● shirt(rgb(119,136,153), army_tshirt).
● shirt(rgb(188,143,143), periwinkle_blouse).
○ shirt(rgb(255,245,238), fashion_cream_blouse).
```

Without any colour constraints, the following query would return all the shirts in the wardrobe:

```
?- [], shirt(Description, Colour).
```

Instead, by defining a *target colour* in the goal via labelled variables, the query can be refined in order to get only those dresses whose *dress_colour* is “similar enough” to the target—with similarity embedded through a suitably-defined combining function f_L.

In our example, two colours are considered as similar if their distance is below a given threshold. Thus, during the unification of labelled variables, if the *dress_colour* is similar to the *target_colour*, the returned label is *dress_colour* (that is, the colour of the selected shirt); otherwise, the empty label is returned, so unification fails.

As a first step, we assume that a colour c is represented as RGB ($c = rgb(r, g, b)$, the threshold is ≤ 30, and the colour distance is normalised and computed as a distance in a 3D Euclidean space. For instance, let us look for all the shirts similar to the papaya colour ○ through the following query, where `Colour` is labelled according to the papaya RGB triple (255, 239, 213):

```
?- Colour^[rgb(255,239,213)], shirt(Description, Colour).

yes. Description / my_pink_blouse, Colour / rgb(255,240,245)
Colour^[rgb(255,239,213)];

yes. Description / old_yellow_tshirt, Colour / rgb(255,222,173)
Colour^[rgb(255,239,213)];

yes. Description / fashion_cream_blouse, Colour / rgb(255,245,238)
Colour^[rgb(255,239,213)]
```

since the normalised distances d_N are:

$d_N($○ $papaya = rgb(255, 239, 213),$ ○ $lightpink = rgb(255, 240, 245))$ = $\boxed{7.25}$
$d_N($○ $papaya = rgb(255, 239, 213),$ ○ $lightyellow = rgb(255, 222, 173))$ = $\boxed{9.84}$
$d_N($○ $papaya = rgb(255, 239, 213),$ ● $armyblue = rgb(119, 136, 153))$ = 40.95
$d_N($○ $papaya = rgb(255, 239, 213),$ ● $periwinkle = rgb(188, 143, 143))$ = 30.88
$d_N($○ $papaya = rgb(255, 239, 213),$ ○ $creamwhite = rgb(255, 245, 238))$ = $\boxed{5.82}$

Making one step further, by adding to the label the neighbourhood information (i.e., the admissible threshold), allows the user to dynamically change the similarity criterion. For instance, the same query as the one in Listing 5.4.2 could be expressed as:

```
?- Colour^[rgb(255,239,213), d = 30], shirt(Description, Colour).
```

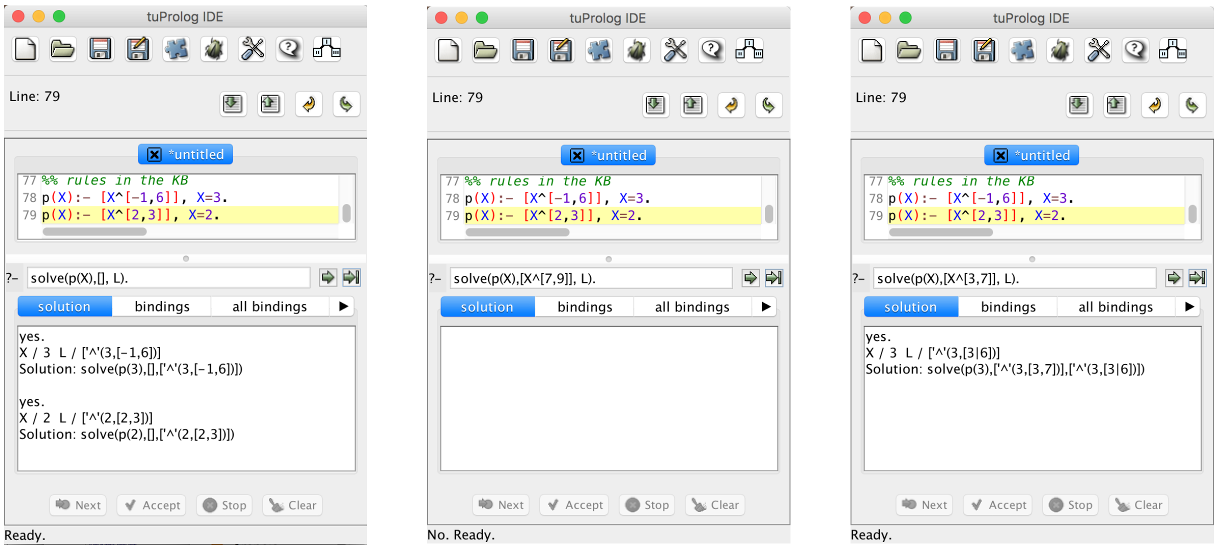
whereas a stricter constraint could be imposed by the following query:

```
?- Colour^[rgb(255,239,213), d = 6], shirt(Description, Colour).

yes. Description / fashion_cream_blouse, Colour / rgb(255,245,238)
Colour^[rgb(255,239,213), d = 6]
```

Once again, while LP is left untouched, LVLP captures a parallel computation on the domain of interest, which affects the final result.

```
%% label_generate(+L1, +L2, -L3) embedding f_L behaviour for integer intervals
label_generate([H1|([T1|T11])], [H2|([T2|T22])], [H3|T3]):-
  ((H1>H2) -> H3=H1; H3=H2), ((T1<T2) -> T3=T1; T3=T2), H3<T3.
```



```
%% rules in the KB
p(X):- [X^[-1,6]], X=3.
p(X):- [X^[2,3]], X=2.
```

```
?- solve(p(X),[], L).

yes.
X / 3  L / ['^'(3,[-1,6])]
Solution: solve(p(3),[],['^'(3,[-1,6])])

yes.
X / 2  L / ['^'(2,[2,3])]
Solution: solve(p(2),[],['^'(2,[2,3])])

Ready.
```

```
?- solve(p(X),[X^[7,9]], L).

No. Ready.
```

```
?- solve(p(X),[X^[3,7]], L).

yes.
X / 3  L / ['^'(3,[3|6])]
Solution: solve(p(3),['^'(3,[3,7])],['^'(3,[3|6])])

Ready.
```

Figure 5.2: *`label_generate/3` example: numeric interval intersection*

Integer intervals

All standard domains for logic languages – including the CLP ones [Coh90] – are also supported. For instance, labels could be used to represent the integer interval over which the logic variable values span: accordingly, the label syntax could take the form `X^[`*min*`,`*max*`]`, and a simple interval program could look as follows:

```
    interval(X):- X^[-1,4].
    interval(X):- X^[6,10].
```

The unification of two variables labelled with an interval would then result in a variable labelled with the intersection of the intervals. Accordingly, the following simple query generates two solutions:

```
    ?- X^[2,7], interval(X).

    yes.
    X^[2,4] ;

    yes.
    X^[6,7]
```

However, the expressiveness of LVLP makes it possible to easily move from the domain of integer intervals to more articulated domains, thus going beyond the reach of constraint logic languages.

For instance, the above example could be easily extended to the domain of *integers with a neighbourhood*, as in the following program:

```
neighbourhood(X):- X^[-1,4], X=3.
neighbourhood(X):- X^[6,10], X=8.
```

There, constant values unify with labelled variables if they belong to the interval in the label. Accordingly, the following query would result in just one solution:

```
?- X^[2,7], neighbourhood(X).

yes.  X / 3
X^[2,4]
```

because the second clause would set `X` out of the interval in the query.

5.5 Remarks & Outlook

The primary results of this Chapter is the definition of the LVLP theoretical framework, where different domain-specific computational models can be expressed via labelled variables, capturing suitably-tailored labelled models. The framework is aimed at extending LP to face the challenges of today pervasive systems, by providing the models and technologies required to effectively support distributed situated intelligence, while preserving the features of declarative programming.

We present the fixpoint and operational semantics, discuss correctness, completeness, and equivalence, and test the effectiveness of our approach through some case studies.

While the first LVLP prototype [CDO15] is currently implemented over `tu`Prolog [DOR05] via the described meta-interpreter, the full integration of the LVLP model in the `tu`Prolog code is currently in advanced stage of development.

The next stage is represented by the design and implementation of a full-fledged logic-based *middleware* for LVLP, which could be exploited to test the effectiveness of LVLP in real-world pervasive intelligence scenarios. As far as the formal aspects are concerned, future work will be devoted to deeper exploration and better understanding of the consequences of applying labels to formulas, as suggested by Gabbay [Gab96]. Other research lines will possibly include the application of the LVLP framework to different scenarios and approaches—such as probabilistic LP [SAFP15], the many CLP approaches [Coh90], distributed ASP reasoning [DP09], and action languages [DFP13].

Part III

Applications of micro-intelligence LP models IoT

In the third part of this thesis, the effectiveness of the integration of all the different, but complementary, contributions presented so far (Part I an Part II) is discussed. In particular, some experiments are presented, showing how LPaaS and LVLP can provide distributed situated micro-intelligence in Smart Environments.

First of all the technology is presented, the current state of `tu`Prolog for IoT technology is provided, that is, a report on the extent to which the `tu`Prolog engine has been actually implemented and extended implementing LPaaS and LVLP models –Chapter 6.

Then, we test the effectiveness of the `tu`Prolog technology in a Smart Home context –namely Home Manager– developed following the Butlers for Smart Spaces approach. The Butlers for Smart Spaces approach is described in Chapter 7, presenting a technology-neutral reference framework focused on users' situatedness and interaction aspects in the IoT environments. Then, some experiments built on the Home Manager Framework (inspired to the Butlers for Smart Spaces approach) are discussed, designed mostly to prove the feasibility and effectiveness of our approach –Chapter 8.

Chapter 6

The tuProlog engine for the IoT

In this chapter we present the tuProlog engine, starting from its key features and moving to the envisioned architecture in a complex IoT system.

Accordingly, Section 6.1 overviews the basic elements and structure of the tuProlog engine, while Section 6.2 describes an envisioned full-fledged micro-intelligence ecosystem exploiting the tuProlog engine with the extension of LPaaS and LVLP, whose building blocks are currently under development. Prototype version of these extensions are available from `http://tuprolog.apice.unibo.it`.

6.1 The tuProlog in a nutshell

tuProlog is a light-weight Prolog system for distributed applications and infrastructures, intentionally designed around a minimal core, to be either statically or dynamically configured by loading/unloading libraries of predicates. tuProlog natively supports multi-paradigm programming, providing a clean, seamless integration model between Prolog and mainstream object-oriented languages – namely Java, for tuProlog Java version, and any .NET-based language (C#, F#..), for tuProlog .NET version.

tuProlog and related packages are released under the GNU Lesser General Public License, via Bitbucket (`https://bitbucket.org/tuprologteam/tuprolog/overview`) as a Git (`http://apice.unibo.it/xwiki/bin/view/TuCSoN/Git`) repository and on Maven Central Repository.

While they all share the same core and libraries, the latter features an *ad hoc* library which extends the multi-paradigm approach to virtually any language available on the .NET platform.

Unlike most Prolog programming environments, aimed at providing a very efficient (yet monolithic) stand-alone Prolog system, tuProlog is explicitly designed to be *minimal*, dynamically *configurable*, straightforwardly *integrated* with Java and .NET so as to naturally support multi-paradigm/multi-language programming (MPP), and *easily deployable*.

Minimality means that its core contains only the Prolog engine essentials – roughly speaking, the resolution engine and some related basic mechanisms: any other feature is implemented in *libraries*. So, each user can customize his/her prolog system to fit his/her own needs, and no more: this is what we mean by tuProlog *configurability*—the necessary counterpart of minimality.

Libraries provide packages of predicates, functors and operators, and can be loaded and unloaded in a tuProlog engine both statically and dynamically. Several standard libraries are included in the tuProlog distribution, and are loaded by default in the standard tuProlog configuration; however, users can easily develop their own libraries either in several ways – just pure Prolog, just pure Java[1], or a mix of the two.

Multi-paradigm programming is another key feature of tuProlog. In fact, the tuProlog design was intentionally calibrated from the early stages to support a straightforward, pervasive, multi-language/multi-paradigm integration, so as to enable users to:

- using any Java[2] class, library, object *directly from the Prolog code* with no need of pre-declarations, awkward syntax, etc., with full support of parameter passing from the two worlds, yet leaving the two languages and computational models totally separate so as to preserve *a priori* their own semantics—thus bringing the power of the object-oriented platform to the Prolog world for free;
- using any Prolog engine *directly from the Java/.NET code* as one would do with any other Java libraries/.NET assemblies, again with full support of parameter passing from the two worlds in a non-intrusive, simple way that does not alter any semantics—thus bringing the power of logic programming into virtually *any* Java/.NET application;
- augmenting Prolog by defining new libraries either in Prolog, or in the object-oriented language of the selected platform (again, with a straightforward, easy-to-use approach based on reflection which avoids any pre-declaration, language-to-language mapping, etc), or in a mix of both;
- augmenting Java[3] by defining new Java methods in Prolog (the so-called 'P@J' framework), which exploits reflection and type inference to provide the user with an easy-to-use way to implement Java methods declaratively.

Last but not least, *easy deployability* means that the installation requirements are minimal, and that the installation procedure is in most cases[4] as simple as copying one

[1]The .NET version of tuProlog supports other languages available on the .NET platform: more on this topic in `http://tuprolog.apice.unibo.it`

[2]For the .NET version: any .NET class, library, object, etc.

[3]This feature is currently available only in the Java version: a suitable extension to the .NET platform is under study.

[4]Exceptions are the Eclipse plugin and the Android versions, which need to be installed as required by the hosting platforms.

archive to the desired folder. Coherently, a Java-based installation requires only a suitable Java Virtual Machine, and 'installing' is just copying a single JAR file somewhere. Of course, other components can be added (documentation, extra libraries, sources..), but are not necessary for everyday use. The file size is quite similar for the Android platform – the single APK archive is 234KB – although an Android-compliant install is performed due to Android requirements. The install process is also quite the same on the .NET platform, although the files are slightly larger. The Eclipse platform also requires a different procedure, since plugin installation have to conform to the requirements of the Eclipse plugin manager: consequently, an update site was set up, where the tuProlog plugin is available as an Eclipse feature. Due to these constraints, file size increases to 1.5MB.

Finally, tuProlog also supports *interoperability* with both Internet standard patterns (such as TCP/IP, RMI) and coordination models and languages. The latter aspect, in particular, is currently developed in the context of the TuCSoN coordination infrastructure [OZ99a, OD01b], which provides logic-based, programmable tuple spaces (called *tuple centres*) as the coordination media for distributed processes and agents.An alternative infrastructure, LuCe [DO01], developed the same approach in a location-unaware fashion: this infrastructure is currently no longer supported.

6.1.1 Predicate categories

In tuProlog, predicates are organized into three different categories:

built-in predicates — *Built-in predicates* are so-called because they are defined at the tuProlog core level. They constitute a small but essential set of predicates, that any tuProlog engine can count on. Any modification possibly made to the engine before or during execution will never affect the number and properties of these predicates.

library predicates — Predicates loaded in a tuProlog engine by means of a tuProlog library are called *library predicates.* Since libraries can be loaded and unloaded in tuProlog engines both at the system start-up, or dynamically at run time, the set of the library predicates of a tuProlog engine is not fixed, but can change from engine to engine, as well as at different times for the same engine. Library predicates cannot be individually retracted: to remove an undesired library predicate from the engine, the whole library containing that predicate needs to be unloaded.

Library predicates can be overridden by theory predicates –that is, predicates defined in the user theory.

theory predicates — Predicates loaded in a tuProlog engine by means of a tuProlog theory are called *theory predicates.* Like above, the set of the theory predicates of a tuProlog engine is not fixed, and can change from engine to engine, as well as at different times for the same engine.

Although library and theory predicates they may seem similar, they are not the same, and are handled differently by the `tu`Prolog engine. The difference between the two categories is both conceptual and structural.

Conceptually speaking, theory predicates should be used to axiomatically represent domain knowledge at the time the proof is performed, while library predicates should be used to represent what is required (procedural knowledge, utility predicates) in order to actually and effectively perform proofs in the domain of interest. So, from this viewpoint, library predicates are devoted to represent more "stable" knowledge than theory predicates. Correspondingly, library and theory predicates are represented differently at run-time, and are handled differently by the engine—in particular, with respect to the observation level for monitoring and debugging purposes. In particular, library predicates are usually step over during debugging, coherently with their more stable (and expectedly well-tested) nature, while theory predicates are step into in a detailed way during the controlled execution. This is also why all the tools in the `tu`Prolog GUI show in a separate way the theory predicates, on the one hand, and the loaded libraries and predicates, on the other.

6.1.2 Engine configurability

`tu`Prolog engines provide four levels of configurability:

Libraries — At the first level, each `tu`Prolog engine can be dynamically extended by loading or unloading libraries. Each library can provide a specific set of predicates, functors, and a related theory, which also allows new flags and operators to be defined. Libraries can be either pre-defined or user-defined. A library can be loaded by means of the predicate `load_library` (Prolog side), or by means of the method `loadLibrary` of the `tu`Prolog engine (Java/.NET side).

Directives — At the second level, directives can be given by means of the `:-/1` predicate, which is natively supported by the engine, and can be used to configure and use a `tu`Prolog engine (`set_prolog_flag/1`, `load_library/1`, `include/1`, `solve/1`), format and syntax of read-terms (`op/3`).

Flags — At the third level, `tu`Prolog supports the dynamic definition of flags to describe relevant aspects of libraries, predicates and evaluable functors. A flag is identified by a name (an alphanumeric atom), a list of possible values, a default value, and a boolean value specifying if the flag value can be modified. Dynamically, a flag value can be changed (if modifiable) with a new value included in the list of possible values.

Theories — The fourth level of configurability is given by theories: a theory is a text consisting of a sequence of clauses and/or directives. Clauses and directives are terminated by a dot, and are separated by a whitespace character. Theories can be loaded or unloaded by means of suitable library predicates.

6.1.3 Libraries

The tuProlog engine is by design choice a minimal, purely-inferential core, which includes only the small set of *built-in*s introduced in the previous Chapter. Any other piece of functionality, in the form of predicates, functors, flags and operators, is delivered by *libraries*, which can be loaded and unloaded to/from the engine at any time: each library can provide a set of predicates, functors and a related theory, which can be used to define new flags and operators.

The dynamic loading of libraries can be exploited, for instance, to bound the availability of some functionalities to a specific use context, as in the following example:

```
% println/1 is defined in ExampleLibrary
run_test(Test, Result) :- run(Test, Result),
                          load_library('ExampleLibrary'),
                          println(Result),
                          unload_library(ExampleLibrary').
```

The tuProlog distribution include several standard libraries, some of which are loaded by default into any engine–although it is always possible both to create an engine with no pre-loaded libraries, and to create an engine with different (possibly user-defined or third party) pre-loaded libraries.

The fundamental libraries, loaded by default, are the following:

BasicLibrary (class `alice.tuprolog.lib.BasicLibrary`) — provides the most common Prolog predicates, functors, and operators. To clearly separate computation and interaction aspects, no I/O predicates are included.

ISOLibrary (class `alice.tuprolog.lib.ISOLibrary`) — provides predicates and functors that are part of the built-in section in the ISO standard, and are not provided as built-ins or by BasicLibrary.

IOLibrary (class `alice.tuprolog.lib.IOLibrary`) — provides the classic Prolog I/O predicates, except for the ISO-I/O ones.

OOLibrary (class `alice.tuprolog.lib.OOLibrary`) — provides predicates and functors to support multi-paradigm programming between Prolog and Java, enabling a complete yet easy access to the object-oriented world of Java from tuProlog: features include the creation and access of both existing and new objects, classes, and resources.

ThreadLibrary (class `alice.tuprolog.lib.ThreadLibrary`) — provides primitives for explicit multi-thread handling.

Other libraries included in the standard tuProlog distribution, but not loaded by default, are the following:

ISOIOLibrary (class `alice.tuprolog.lib.ISOIOLibrary`) — extends the above IOLibrary by adding ISO-compliant I/O predicates.

SocketLibrary (class `alice.tuprolog.lib.SocketLibrary`) — provides support for TCP and UDP sockets.

DCGLibrary (class `alice.tuprolog.lib.DCGLibrary`) — provides support for Definite Clause Grammar, an extension of context free grammars used for describing natural and formal languages.

Further libraries exist that are *not* included in the standard tuProlog distribution, because of their very specific domain: they can be downloaded from the tuProlog site, along with their documentation.

6.1.4 Multi-paradigm programming

tuProlog supports multi-paradigm and multi-language programming between Prolog and Java in a complete, four-dimensional way:

- using Java from Prolog: *OOLibrary*
- using Prolog from Java: *the Java API*
- augmenting Prolog via Java: *developing new libraries*
- augmenting Java via Prolog: *the P@J framework*

Using Java from Prolog: *OOLibrary*

The first MPP dimension offered by tuProlog is the ability to fully access objects, classes, methods, etc. in a full-fledged yet straightforward way, completely avoiding the intricacies (object and method pre-declarations in some awkward syntax, pre-compilations, etc) that are often found in other Prolog systems. The unique tuProlog approach keeps the two computational models clearly separate, so that neither the Prolog nor the Java semantics is affected by the coexistence of the logical and imperative/object-oriented paradigms in the same program. In this way, any Java package, library, etc. is immediately available to the Prolog world with no effort, So, for instance, Swing classes can be easily exploited to build the graphical support of a Prolog program, and the same holds for JDBC to access databases, for the socket package to provide network access, for RMI to access remote Java objects, and so on.

The two basic bricks of OOLibrary are:

- the mapping between Java types and suitable Prolog types;
- the set of predicates to perform operations on Java objects.

Using Prolog from Java: *the Java API*

The tuProlog Java API provides a complete support for exploiting Prolog engines from Java: its only requirement is the presence of `tuprolog.jar` (or the more complete `2p.jar`) in the Java project's class path. The API defines a namespace (`alice.tuprolog`) and classes to enable the definition in Java of suitable objects representing Prolog entities (terms, atoms, lists, variables, numbers, etc, but also Prolog engines, libraries and theories), and use them to submit queries and get the results back in Java, thus effectively supporting multi-paradigm, multi-language programming.

Augmenting Prolog via Java: developing new libraries

So far, the two first dimensions of tuProlog's support to multi-paradigm, multi-language programming have been explored, that enable a language (and the corresponding paradigm) to be used from the other. The two further dimensions concerns *augmenting* the language instead—that is, exploiting a language (and a paradigm) to increase the other.

In this section the focus is on augmenting Prolog from Java, exploiting the latter to increase the first by developing new tuProlog libraries; the next Section (6.1.4) will focus on the opposite direction, exploiting Prolog to augment Java via the so-called *P@J* framework.

Moreover, although tuProlog libraries are expressed in Java, they are not required to be fully implemented in this language. In fact, Java-only libraries are the simplest case, but hybrid Java + Prolog libraries are also possible, where a Prolog theory is embedded into a Java string so that the two parts cooperate to define the overall library behavior. This opens further interesting perspectives, that will be discussed below.

Augmenting Java via Prolog: the P@J framework

The last dimensions of tuProlog's support to multi-paradigm, multi-language programming is still a form of *augmenting* a language (that is, exploiting a language and a paradigm to increase the other)—in this case, augmenting Java from Prolog, exploiting the so-called *P@J* framework [CV08].

This approach makes it possible to "inline intelligence" into Java code, enabling Prolog to be used for implementing Java (abstract) methods, via Java reflection and suitable annotations. The basic idea is that the methods to be implemented in Prolog are declared `abstract` from the Java syntax viewpoint[5], so that the Java compiler does not expect to find any implementation, while annotating them with the Prolog clauses that provide the actual implementations. On the user side, the factory method `PJ.newInstance` will be used to automatically create a Java implementation of this method, which interacts with the Prolog engine in a totally transparent way.

6.1.5 JSON Serialisability

Version 3.2.0 of tuProlog introduces the support for the serialisation and deserialisation of the engine's state via JSON representation [JSO17]. This new feature enables the user to serialise the engine and re-create it on another node.

Note that, this is a quite heavy action because it needs to serialise the engine's state, to send it to another node, with possible network overhead, and to re-compute the solution on the destination node. For such a reason, the tuProlog engine's state can be of two different types: *FullState* or *ReducedState*. The first is composed by the knowledge base of the engine, the last computed query, the number of result given to the user and the timestamp of the serialisation. The latter is equal to the first, except for the knowledge base, which is not serialised.

Following this approach, the engine answers the IoT demand for interoperability, both with a light way primitive –*ReducedState*– and with a full fledged primitive –*FullState*. Accordingly, depending on the requirements of the scenarios the user can exploit the more appropriate primitive. For instance, the *ReducedState* can be exploited in case of necessity of stateful connection between the client and the engine, while the *FullState* can be exploited for mobility or recovering of a node.

The *JSON API* is built on top of Gson [GSO], a tool developed by Google which automates serialisation and deserialisation of POJO [6] [POJ17]. So, in order to use the tuProlog *JSON API*, Java users have to add the Gson jar [GSO] to the dependency of their project. Users can find the `gson-2.6.2.jar` file into the Java distribution of tuProlog. For .NET users, the `tuprolog.dll` already contains the Gson core, so no other libraries are required.

6.2 The tuProlog under the IoT vision

6.2.1 Related & Motivation

There are many frameworks that make use of Prolog to realise IoT platforms.

Prolog is well suited for developing intelligent solutions due to its inherent features which include unification, resolution and depth first search. Its declarative nature and

[5]Of course, the corresponding class must be syntactically qualified `abstract`, too.

[6]In software engineering, a plain old Java object (POJO) is an ordinary Java object, not bound by any special restriction and not requiring any class path.

efficient handling of tree structures makes it highly efficient and productive. Prolog also offers dynamic code, which allows traits such as learning and intelligence to be easily incorporated. Morover, the ability of a Prolog engine and the parallelism exhibited by mobile agents can be exploited to allow emulation of complex algorithms.

Jini [Tar99] and Movilog [ZCM03] reportedly use a combined Java and Prolog based framework for creating agent based systems. The use of two different languages generally deters a developer from using such platforms as it makes the knowledge of these concerned languages mandatory. A pure Prolog based mobile agent framework possibly could serve the purpose better. ALBA [DCT07] and IMAGO [Li01] cater to a Prolog- only environment for mobile agent based system development. ALBA is more of a library rather than a platform, and uses SICStus Prolog [CF14] for commissioning agents. IMAGO [DCT07], on the other hand, is a hierarchy-based mobile agent system: agents in IMAGO are categorised into three types - Stationary imagos which are powerful but lack mobility, Worker imagos which are mobile but limited in ability, and Messenger imagos which are used for communications between different imagos. However, IMAGO comes as a separate package and cannot be used in tandem with other interpreters. Typhon is a mobile agent framework which uses LPA Prolog (http://www.lpa.co.uk). Typhon (a mythical monster) assisted by the Chimera (the offspring of Typhon) Agent System provided with LPA Prolog facilitates an environment to create and program mobile agents with logic embedded within.

Between the other, our choice shakes down to the tuProlog system. Developing an IoT Prolog framework exploiting tuProlog would mean that these all the Prolog engines envisioned into the system can benefit from the key features of the tuProlog system, particularly suitable in IoT architectures:

- minimal and configurable engines running without heavy computational costs [DOR01]
- multi-platform technology, including resource-constrained devices [DOC13]
- interoperability with widespread languages, technologies, and standards
- clean model integration for conceptual integrity [DOR05]

Moreover, tuProlog is a light-weight Prolog system for distributed applications and infrastructures, intentionally designed around a minimal core; it can be either statically or dynamically configured by loading/unloading libraries of predicates. It natively supports multi-paradigm programming [Denti et al., 2005], providing a clean, seamless integration model between Prolog and mainstream object-oriented languages and finally runs on most known platforms and devices. Agents can easily make use of extra features required to exhibit intelligence just loading a library. The tuProlog system allows a user to exploit the intelligence programming abilities inherent in Prolog and at the same time program and rapidly emulate complex parallel algorithms.

6.2.2 tuProlog LPaaS

The LPaaS prototype available on Bitbucket [tup01] has been implemented on top of the tuProlog system, which provides not only a light-weight engine, particularly well-suited for this kind of applications, but also a multi-paradigm and multi-language working environment, paving the way towards further forms of interaction and expressiveness.

In particular, from the service perspective, there are key requirement, captured natively from the tuProlog engine:

- ensuring the interoperability –natively supported by JSON serialisation
- exploitability by applications as a *library service*—that is, from a software engineering standpoint, a suitably-encapsulated collection of related functionalities

From the MAS perspective, we choose tuProlog because of its peculiar blend of imperative, object-oriented, and logic programming styles: apart from being Java-based, light-weight, and easy deployable, it also enables and promotes a *multi-paradigm* programming style, where the Prolog code can invoke Java code and viceversa, yet keeping the two computational models clearly separate [DOR05].

Details on the RESTful and the MAS-LPaaS prototype implementations can be found in Section 3.4.

6.2.3 tuProlog LVLP

The operational semantics of LVLP is captured by a meta-interpreter developed on the top of tuProlog.

Once again, our choice follows the tuProlog design that inherently enables the injection of Prolog programs within pervasive systems, as well as their integration with diverse programming languages and paradigms, over computing platforms of any sort.

While the first LVLP prototype [CDO15] is currently implemented over tuProlog [DOR05] via a described meta-interpreter, the full integration of the LVLP model in the tuProlog code is currently in advanced stage of development.

Details on the Labelled-tuProlog prototype implementations can be found in Section 5.4.

Chapter 7

The Butlers for Smart Space Reference Architecture

In this Chapter, we explore the construction of Smart Environments (SE) exploiting the micro-intelligence models proposed in Part I and Part II of these thesis.

In recent years a plethora of novel *open*, *pervasive*, highly *dynamic*, and mostly *unpredictable* systems, such as self-organising ones, have emerged, presenting brand new challenges demanding for innovative models of intelligence [Mar16a].

Smart Environments, and in particular *Socio-Technical Systems* (STS) and *Knowledge-Intensive Environments* (KIE), are both examples of systems conceived and designed to combine business processes, technologies and people's skills [Whi06] to store, handle, and make accessible large repositories of information [Bha01]. Recent advances in sensor networks, micro-scale, and embedded vision technologies are enabling the vision of *ambient intelligence* and *smart environments*. Environments have to react in an attentive, adaptive and active way to the presence and activities of humans and other objects in order to provide intelligent and smart services. Heterogeneous sensor networks with embedded vision, acoustic, and other sensor modalities offer a pivotal technology solution for realising such systems. Networks with multiple sensing modalities are gaining popularity in different fields because they can support multiple applications that may require diverse resources. Here, the Internet of Things plays a key role: appliances and devices of any sort are networked together and can possibly embed some form of (limited) intelligence to provide suggestions from the user's context and habits.

Managing their *distributed intelligence* is of paramount importance, for guaranteeing their functionalities, as well as for providing desirable non-functional properties [EM14]—e.g., scalability, fault-tolerance, self-* properties in general. Modern autonomic component-based systems [DYZ09] aim to change their configuration during the running time in order to satisfy the user requirements or the environment changes. Usually, the reconfiguration is done after analysing the requirements and making the suitable decision, exploiting intelligence mechanism [DDF^{+}06]. Moreover, an important component of an intelligent environment is to anticipate actions of a human inhabitant and then automate them. However, engineering and design intelligence is far from trivial, mostly due to the peculiar characteristics of the aforementioned systems.

The *micro-intelligence* approach, such as LPaaS and LVLP, aims at modelling intelligence in such systems by properly supplying inference mechanism for knowledge-intensive STS.

In this chapter we introduce a framework, namely the *Butlers for Smart Spaces*, conceived starting from the micro-intelligence notion for the design and development of STS and KIE —and more in general of SE. In particular, in Section 7.1 we address the intelligence issues of SE, making an overview of the state of the art; then we present the *Butlers* vision [Den14] and the *Butlers for Smart Spaces* framework [CD16a, CD16b], as a conceptual reference framework for the design and development of advanced services to users immersed and situated in time and space Section 7.3.

7.1 Micro-Intelligence in Smart Environment

As mentioned in [CD16b], Smart Environments (also called Smart Spaces) [CFJ+04b, WDC+04] aim at augmenting apartments, offices, museums, hospitals, schools, malls, universities, outdoor areas with smart objects and systems: both people and smart objects are immersed in time and space, and computer systems seamlessly integrate into people's everyday lives “anywhere, anytime” [SM03]. Intelligence pervades the environment, and space and time awareness sets the base for a contextualised, adaptive user experience.

A key aspect of these scenarios is that the technology complexity is amplified by the organisational complexity of the application domain: such systems need to be conceived, designed and developed taking into account both the technological and the human/organisational aspects from the earliest stage, combining different dimensions and behaviour from *pervasive*, *distributed*, *situated* and *intelligent* computing—altogether [RPTC15]. Because of their characteristics, Smart Environments assume the co-existence of heterogeneous entities, differing in terms of execution platform, development language, enabling technologies, support infrastructures, models, roles, objectives, etc.—which is why they inherently call for a *multi-paradigm* approach: in fact, their development calls for skills, concepts, methodologies, technologies from a variety of fields (such as, for instance, AI, coordination, distributed systems, organisational sciences, etc.).

The Internet of Things [Mic15a, RPTC15, SKP+11] is providing the fundamental bricks to make such scenarios concrete: appliances and devices of any sort are networked together and can possibly embed some form of (limited) intelligence to provide suggestions from the user's context and habits (examples include Apple [App14], Amazon [Ama16], Google [Goo14a], Microsoft [Mic15a], Samsung [Sam15], to name just a few).

Yet, most applications today merely provide some nice app, very much like a novel form of remote-control–which is probably the main reason for the gadget-like feeling that still affects most of the available prototypes [Ama16, App14, Goo14a, Mic15a, Sam15]. To make one step further, three key preconditions seem to be *(i)* the availability of an effective coordination middleware, going beyond the basic support to interoperability; *(ii)* an effective support to *distributed situated intelligence*, intended as awareness of the environment and chance to react to changes—the basic brick to support contextualised reasoning and possibly prediction; and *(iii)* guidelines and enabling techniques exploiting skills, concepts, methodologies, technologies from the most diverse (socio-technical) fields, in a multi-paradigm perspective.

In this context, the goal is to define an intelligent system which is able to anticipate, predict, and take decisions with autonomy. The key features the system needs to manage, related to micro-intelligence issue, can be grouped into the following ones:

- modelling of context (intended as time and space) and context reasoning –i.e. situatedness;
- proactive recommendation;
- situations prediction.

In the following we discuss these features along with the main research works in the area.

7.2 Related Works

Research addressing Smart Environments has in the past largely focused on network- and hardware-oriented solutions [AN06]. AI-based techniques which promote intelligent behaviour have not been examined to the same extent, until the last decades. Then, there has been a growing body of research on the use of context-awareness as a technique for developing pervasive computing applications that are flexible, adaptable, and capable of acting autonomously on behalf of users.

7.2.1 Context Reasoning

A large part of the research investigates approaches to modelling context information used by context-aware applications and reasoning techniques for context information. Context awareness [ADB$^+$99], as a core feature of ubiquitous and pervasive computing systems, has existed and been employed since the early 1990s.

Reasoning about space and time plays an essential role in our everyday lives: when navigating around our cities through road networks, when working according to schedules at our work places, or when finding our way around at home. In the latter case, smart environments can support us in this reasoning process, as they can remind us to do certain things at certain times or to be at certain places at certain times. This is of particular benefit for the elderly or people with cognitive impairments.

Due to the inherent complexity of context-aware applications, the development should be supported by adequate software engineering methods. The overall goal is to develop evolvable context-aware applications. Therefore the design of the general functions of such applications should not be intertwined with the definition and evaluation of context information, which is often subject to change.

Context, as intended in this work, can be considered as a specific kind of knowledge. Thus, it is quite natural to investigate if any known framework for knowledge representation and reasoning may be appropriate for handling context.

A number of context modelling and reasoning approaches have been developed over the last decade ranging from very simple early models to the current state-of-the-art context models. The research on the models was accompanied by development of context management systems that were able to gather, manage, evaluate and disseminate context information [BBH+10]. Three of the main research approaches, which influenced the Butlers reference architecture, are discussed.

Context Modelling Language *Fact-based context modelling* [BBH+10] approaches have been investigated, including the object-role modelling approach, originated from attempts to create sufficiently formal models of context to support query processing and reasoning, as well as to provide modelling constructs suitable for use in software engineering tasks such as analysis and design. Early context modelling approaches, such as attribute-value pairs, could not satisfy these requirements, particularly as the types of context information used by applications grew more sophisticated. Context modelling approaches have their early roots in database modelling techniques. In particular, it focuses on the Context Modelling Language (CML), which was described in a preliminary form by Henricksen et al. in 2002 [HIR02] and refined in later publications [HI04, HI06]. CML is based on Object-Role Modeling (ORM) [HM10], which was developed for conceptual modelling of databases. The formality of ORM and the CML extensions makes it possible to support a straightforward mapping from a CML- based context model to a runtime context management system that can be populated with context facts and queried by context-aware applications. A detailed discussion of both the model and the software engineering process used in conjunction with CML can be found in [HI06].

Ontology-based models of context information The tradeoff between expressiveness and complexity of reasoning has driven most of the research in symbolic knowledge representation in the last two decades, and description logics [BCM+03] have emerged among other logic-based formalisms, mostly because they provide complete reasoning supported by optimised automatic tools. Since ontologies are essentially descriptions of concepts and their relationships, it is not surprising that the subset of the Web Ontology Language (OWL) [GHM+08] admitting automatic reasoning (i.e., OWL-DL) is indeed a description logic. Ontology-based models of context information exploit the representation and reasoning power of these logics for multiple purposes: *(a)* the expressiveness of the language is used to describe complex context data that cannot be represented, for example, by simple languages like CC/PP [KRW$^+$04]; *(b)* by providing a formal semantics to context data, it becomes possible to share and/or integrate context among different sources; *(c)* the available reasoning tools can be used both to check for consistency of the set of relationships describing a context scenario, and, more importantly, to recognise that a particular set of instances of basic context data and their relationships actually reveals the presence of a more abstract context characterisation (e.g., the user's activity can be automatically recognised). The formalism of choice in ontology-based models of context information is typically OWL-DL [HPSvH03] or some of its variations, since it is becoming a de facto standard in various application domains, and it is supported by a number of reasoning services. OWL-DL ontological models of context have been adopted in several architectures for context-awareness; among the others, we recall the Context Broker Architecture (CoBrA) [CFJ04a] and the SOCAM [GWPZ04] middleware, that adopt the SOUPA and CONON ontologies, respectively.

High-level context abstractions Information from physical sensors, called *low-level context* and acquired without any further interpretation, can be meaningless, trivial, vulnerable to small changes, or uncertain [BP80]. Schilit et al. [SAW94] observed hence that context encompasses more than just the user's location, because other things of interest, including the user's social situation, are also changing. The limitation of low-level contextual cues when modelling human interactions and behaviour risks reducing the usefulness of context-aware applications. A way to alleviate this problem is the derivation of higher-level context information from raw sensor values, called context reasoning and interpretation. The idea is to abstract from low-level context by creating a new model layer that gets the sensor perceptions as input and generates or triggers system actions. In the literature, different notions have been employed to refer to this higher-level context layer. Situational context [GSB02] and situation [Dey01, DY06] are the most common ones. The notion of situation is used as a higher-level concept for a state representation. Initially, the term "situation" was used in linguistics and natural language semantics.

Time & Space Focussing on the following broadly formulated questions *What does "situatedness" mean? How does situatedness influence the concept of knowledge and vice versa —how does knowledge produce situatedness?* many works reflect in deep on the two main concepts around which the situatedness is concived, namely time and space.

Time. The ability of providing and relating the temporal representation of facts at different levels of granularity is an important research theme in computer science and, in particular, in the database area [BJW00, CR04]. In this area, one of the earliest formalisation of the concept of time granularities is described in [CR87]. Other relevant works on time granularities has been done in the context of classical and temporal logics [CCMSP93, BB94] and some of these works have been applied to the verification of real-time system specifications and to temporal reasoning in the artificial intelligence area. The representation and reasoning about temporal constraints related to several granularities is very important also for Smart Environments applications, which are applications related to environments which are capable to react "in an intelligent way" to some occurring situations [Aug05, AN04, GSK+04].

Space. An important context in many context-aware applications. Most context definitions mention space as a vital factor: e.g., Schilit, Adams and Want define three important aspects of context as "Where you are, who you are with and what resources are nearby" [SAW94]. Also, in the most frequently used context definition by Dey et al. [Dey01], space can be seen as a central aspect of context entities. Thus, some context modelling approaches give space and location a preferential treatment. As we will see, space is well suited to organise and efficiently access context information. Spatial existence also serves well as an intuitive metaphor for non-physical context information (e.g., virtual information towers [LKRF99] for context-tagged web pages or Pascoe's Stick-E-Notes [Pas97]). Most spatial context models are fact-based models that organise their context information by physical location. Moreover, the spatial context model developed in the Nexus project (called Augmented World Model [NM01]) is an object-based class hierarchy of context information that supports multi-inheritance, multi-attributes, and both a geometric coordinate system and a simple symbolic location system. The Nexus context model was designed to be sharable between different context-aware applications in a potentially global scope and thus to be scalable to a high amount of context data [GBH+05]. In contrast to the Nexus model, the Equator project context model [MRS04] is a typical contextual ontology that represents all tiers by an OWL class model. Its location model is a hierarchical notion of inter-connected symbolic spaces, such as Buildings, Floors and Rooms. Properties define spatial relations between these spaces. Although the ontology also offers coordinate features (properties that represent, e.g., a GPS location), Millard et al. states that it is very hard to perform any inference over them using a normal reasoner, as they are usually not spatially aware.

7.2.2 Proactive Recommendation

Recommender Systems (RSs) are software tools and techniques providing suggestions for items to be of use to a user [RRS11]. In particular, we are interested in Knowledge-based Recommender Systems [Tre00], which pursue a knowledge-based approach to generate a recommendation, by reasoning about what items meet the user's requirements. Knowledge is built by recording user preferences/choices or through asking users to provide information as to the relevance of the choices. The similarity function represents an estimate of the extent that user needs correlate with available content item options; the similarity function's value is typically shown to illustrate the usefulness of each recommendation.

So, one of the goals of a recommender system is to help the user in finding interesting objects, based on interests, preferences and personal characteristics.

For instance, Yang et al. proposed a location-aware recommender system that accommodates customers' shopping needs with location-dependent vendor offers and promotions [YCD08]. Yuan and Tsao introduced a framework which enables the creation of tailor-made campaigns targeting users according to their location, needs and devices' profile (i.e. contextualised mobile advertising) [YT03].

Such systems perform some similarity measure between object-object, object-user and user-user, with the purpose to determine a set of objects to be recommended to a specific user. The result can be a prediction, i. e., a numeric value given to each object that expresses likeliness of an object for the user; or a recommendation, i. e., a top-N list of objects [Kar05]. Recommendations may be based on similar items to those a given user has liked in the past (content-based recommendation); or on items owned by users whose taste is similar to those of the given user (collaborative recommendation). Combining content-based and collaborative recommendations originate hybrid approaches, which are commonly used, considering that both types of recommendations may complement each other [dCFLHRM10, van05]. Besides the difference given by the above recommendation approaches, recommender systems are also differentiated by [BHC98]: the items they recommend (systems have been developed to recommend web pages [dCFLHRM10], movies [MLdlR03], etc.); the nature of the user profile they use to guide the recommendations (e.g. history of items accessed by the user, topics indicating user interest, etc.); the recommendation techniques (mainly, how the user model is represented, what kinds of relevance mechanisms are used to update the user model, and which algorithm is used to generate recommendations); and the recommendation trigger, i.e. whether the recommendation is started by the user or by the proactive behaviour of the system.

Collaborative filtering recommender systems usually use the whole user profiles (the whole set of rated objects) to calculate similarities among them. In situation where the recommender system should recommend objects in various domains, this may lead to mistakes, once the system assume that if two users have the same preferences in a certain domain, so in others domains their preferences will also resemble. But, in many cases, this is not true. Context-aware platforms may also benefit by the use of recommender systems, mainly to support recommendation of the services offered by the platforms. In this case, the user would not need to manually request the platform for the services in which he is interested. These services would be automatically recommended based on the user's context and profile. It is important to realise that the central goal of context-aware systems and recommender systems is the same, i.e. providing users with relevant information and/or services selected from a potentially overwhelming set of choices [vSPK04]. The difference is that in the former one the selection is based on the user's context while in the latter it relies on the user's interest. This suggest that these kinds of systems are rather complementary than competing, hence motivating their integration.

Between the others, COReS [CF07], is a Context-aware, Ontology-based Recommender system for Service recommendation, which uses the capabilities of the Infraware platform to support services selection, making service offer by this platform more efficient, personalised and proactive, and thus satisfying the needs of the user in a particular context. It is a hybrid system implementing recommendation technics based on an existing system named KARe. An innovation of COReS is the adoption of compartmentalised the user profiles according to different domains. The suitable profile part for a given recommendation is selected in time of prediction, based on the user's context.

One of the main attempt to integrate context awareness with mobile recommender systems is the COMPASS application [vSPK04]. COMPASS, which means Context-aware Mobile Personal ASSistant, is a context-aware mobile tourist application that serves a tourist with information and services based on his interests and current context. The application consists of a map showing the user location. Depending on his profile and goal, the system selects nearby buildings, buddies and other objects and shows (recommend) them on the map and in a list. The application is built upon the WASP platform [Cos03] that provides generic supporting services, such as a context manager and service registry. To show objects on the map, COMPASS first queries the service registry for search services that are bound to deliver objects related to the user's context. The WASP retrieves services matching the user's context and goal. After that, the relevant search services are queried to retrieve the objects matching the context's criteria, e.g. being in a certain distance from the user. The retrieved objects are then sent to the recommendation engine which scores each object based on the user's interests and contextual factors. The resulting objects and scores are displayed on the map and in the list of objects.

Another approach is Ubi-Mate [HB], a mobile recommender system proposed by Annie Chen. Chen proposed a context-aware collaborative filtering system that can predict a user's preference in different context situations based on past user-experiences. The system uses what other like-minded users have done in similar context to predict a user's preference towards an item in the current context [Che05]. The context information that UbiMate include are: user information (profile and rating history), social environment (alone, friends, family, etc.), tasks (activity), location (coordinates), weather (derived from coordinates) and time (time of interaction). The composition of different context information establishes a snapshot of context. The collaborative filtering algorithm the user's profile and a snapshot of the current context along with the rating, and use statistical methods to predict the items the user will most prefer.

7.2.3 Situation Prediction

Predictive Ambient Intelligence (PAI) techniques are used in a smart home environment in order to forecast the behaviour of inhabitant under a monitoring environment [SM15]. A Predictive Ambient Intelligence environment, fon instance, gathers information from Wireless Sensor Networks (WSN) including environmental changes and occupants' interactions with the objects within the monitoring environment. Collected data are used to determine the behaviour of inhabitant at different times by using prediction methods. The prediction involves the extraction of patterns related to sensor activations and then, as in [DC05], it is used to classify the sequence of activities and match it to predict the next activity.

At the heart of any prediction system, a prediction algorithm is essential to function in a dynamic world. For an agent to perform prediction, it should be capable of applying the limited experience of environmental event history to a rapidly changing environment, where event occurrences are related by temporal relations. In particular, a smart space must be able to accurately predict mobility and other activities of its inhabitants [DC05]. Using these predictions, the SS can accurately route messages and multimedia information, and can automate activities that would otherwise be manually performed by the actors. Oftenly, machine learning techniques are required to predict actors movement patterns, tasks, and typical interactions with the space, and to use that information in automating decisions, routing information, and optimising inhabitant comfort, security, and productivity.

Forecasting in smart environments equipped with sensor networks is a learning task. A major task for the intelligent home monitoring system is to have the ability to perceive, understand and realise the new situations. This will support an interpretation of sensory information in order to represent, understand the environment and perform correctly based on the prior knowledge when there is a situational change. For execution of these tasks, a variety of methods such as Analysis of Knowledge Discovery, Soft Computing Techniques and Statistical Modelling methods were introduced.

Analysis of Knowledge Discovery

The purpose of this analysis is an attempt to learn the daily activity patterns from a large data set to realise a new situation for predicting the abnormal behaviour of the system. Some of the existing methods are context-aware case-based reasoning as proposed in [KSL12]. The objective of [KSL12] is on the activity-based context-aware services; the contextual information is presented in terms of environment, temporal and inhabitant identification. The context prediction method is proposed to improve the accuracy of diagnosing a person's state of health with a single activity sensor. The shortcoming of this technique is that when the sample data are incomplete and somewhat inconsistent due to sensor conditions, the reasoning process is not accurate. Similarly, knowledge-driven approaches to activity recognition in a smart home using contextual information with large repositories are proposed [CNW12], while in [GWT$^+$09], methods for recognising sequential, interleaved and concurrent activities using emerging patterns are discussed.

Soft Computing Techniques

Soft computing, as opposed to traditional computing, deals with approximate models and gives solutions to complex real-life problems. Unlike hard computing, soft computing is tolerant of imprecision, uncertainty, partial truth, and approximations. In effect, the role model for soft computing is the human mind. Soft computing is based on techniques such as fuzzy logic, genetic algorithms, artificial neural networks, machine learning, and expert systems [Ibr16]. Study on inhabited intelligent environment, like for instance iDorm [CCC+04], exploit the Adaptive Fuzzy Inference System [QY07] mechanism for prediction in various phases for learning, controlling and adaption. Different techniques of clustering approach and quantification of fuzzy membership functions are used to extract the fuzzy membership function for the collected information on the observation of interactions by the inhabitant. Accordingly, fuzzy rules were extracted from the documented data and a control agent approximates the likelihood of the activity by adjusting with the current situation [DHCL04]. Several methods including neural networks, heuristic and machine learning techniques are used to extract ADL patterns from observed daily activities and these patterns are used later as predictive models [BCR09]. However, these techniques need alternating solution (i.e. the activities learning model need regular updates), if the execution environment is changed and there can be issues of data inadequacy for adapting to a new system. There are methods for deriving abnormal activity models from a general normal model via a Kernel Nonlinear Regression (KNLR) and Support Vector Machines (SVM's) and Hidden Markov Models to reduce the false positive rate in an unsupervised manner, however these methods were applied for wearable sensors on the body [CEF+12].

Probabilistic and Statistical Analysis

In healthcare sector, many methods have been used such as statistical analysis [AMSM08] and probabilistic methods [YKB+03] to predict health issues; most of the prediction models depend on arithmetic formulas and probabilistic models. Arithmetic formulas predict defects in software using size and complexity metrics, testing metrics and processing quality data. The probabilistic models mostly used are multivariate approaches and Bayesian belief networks [FN99].

Probabilistic models are more suitable than the above mentioned two techniques. There are several investigations on activity recognition aiming on the use of probability concepts, and statistical analysis procedures such as [NH12, STF08]. Some of them are Markov representations [NH12] and their variants Conditional Random Fields (CRFs). The methods require large training data for acceptable reasoning with no guarantee for detecting an appropriate abnormality condition.

In order to detect behavioural changes and for better prediction of object usages in a smart home; a new framework termed "wellness determination" in smart home has been devised and extensive work is being performed [SGRM12, MMS+12].

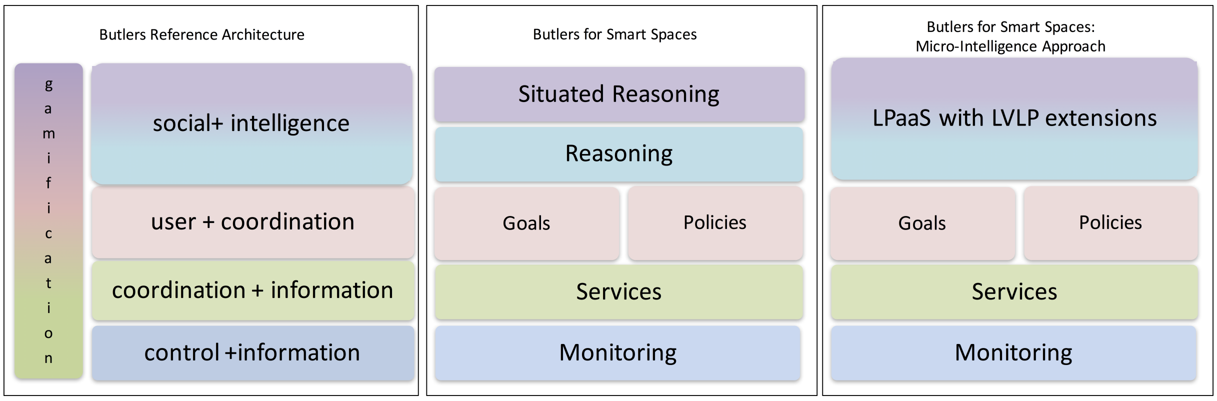


Figure 7.1: *Micro-Intelligence approach in the Butlers Vision*

7.3 Butlers for Smart Space

7.3.1 Towards a Model & Framework for Micro-Intelligence in the Smart Space context

With respect to the requirements detailed above –namely, intelligent system able to anticipate, predict, and take decisions with autonomy– we revised the Butlers architecture [Den14] (a technology-neutral framework made of seven conceptual layers, relating technologies, features and the corresponding value-added for users) in the micro-intelligence vision.

The micro-intelligence definition (chapter 1 Section 1.1) finds its position in the upper level of the Butlers architecture, providing intelligent situated models and mechanisms. Figure 7.1 shows how the two upper layers, *Reasoning* and *Situated reasoning*, are captured by the micro-intelligence vision and, in particular, by the LPaaS and LVLP models.

In this scenario, micro intelligence sources can be provided by exploiting application models and technologies, or infrastructure/middleware services or artefacts for intelligent coordination of the system entities, enabling agents to have a higher-level, domain-free reasoning capability. As mentioned above, the approach results especially effective in Smart Environments where knowledge representation and qualitative spatial and temporal reasoning are required more often.

The micro-intelligence approach, concretised by the LPaaS and LVLP models, is a possible answer to the required key features of SEs discussed in Section 7.1. In particular:

- *modelling of context* is provided both by LPaaS –by means of suitable LP knowledge– and by the LVLP approach that supply a model to overcome LP limitation in the description of domain specific situation and local peculiarity;
- *context reasoning* is provided by LP inference engines which are situated in time and space and can deal with the surrounded environment

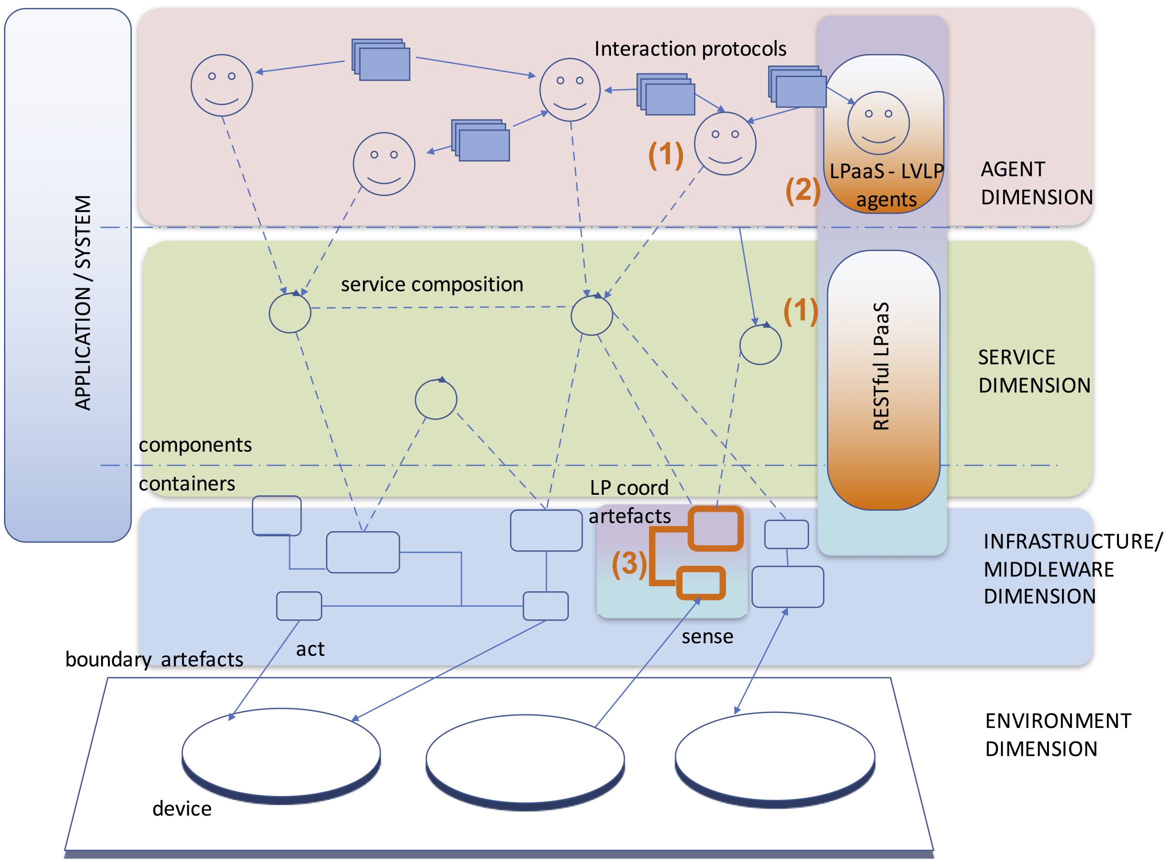


Figure 7.2: *Micro-Intelligence approach in the Butlers Vision: MAS specialisation*

- *proactive recommendations* follows from the two previous items, supplying all the abstractions for autonomous reactions, as a more autonomous approach for recommendation delivery, by anticipating information needs in advance and acting on users' behalf with minimal efforts and without disturbance;
- *prediction* as a consequence of the to combination of history and situated reasoning.

Although this reference architecture is technology neutral the blend of this vision with the Multi Agent System (MAS) architectures can bring additional benefits, especially in the SEs and pervasive IoT scenarios as highlighted in many recent works. Since agents are reactive, proactive and exhibit an intelligent and autonomous behaviour, MAS emerges as a natural approach to develop IoT systems and smart environments [VZ17, SC17, MPS$^+$16, MAT13].

The specialisation of the micro-intelligence approach blended with the Butlers for Smart Spaces vision is depicted in Figure 7.2, where different dimensions and abstractions are taken into consideration. At the bottom layer, the physical / computational environment lives, with *boundary artefacts* [ORV06] taking care of its representation and interactions with the rest of the MAS. Then, typically, some middleware infrastructure provides common API and services to application-level software – i.e. the containers where service components live – there including the *coordination artefacts* [ORV06] governing the interaction space. Finally, on top of the middleware, the application / system as a whole lives, in LPaaS-LVLP MAS view as a mixture of services – possibly RESTful, as for LPaaS as a WS – and agents.

Figure 7.2 illustrates how the Butlers vision, exploiting the LPaaS-LVLP model makes use of an hybrid approach where *(1)* some agents are kept more lightweight and rely on infrastructural services (or other more "intelligent" agents) to get LPaaS-LVLP functionalities, *(2)* some agents embed the LPaaS-LVLP functionalities, and *(3)* some LP functionalities are embedded in some functionalities provided by the middleware (namely by the containers).

The traditional MAS architecture is enriched with the notion of LPaaS-LVLP agent / service, which allows for situated reasoning on locally available data by design.

This model is the basis and the very core of the Home Manager architecture, described in the next Chapter, designed and implemented to show the effectiveness of such an approach.

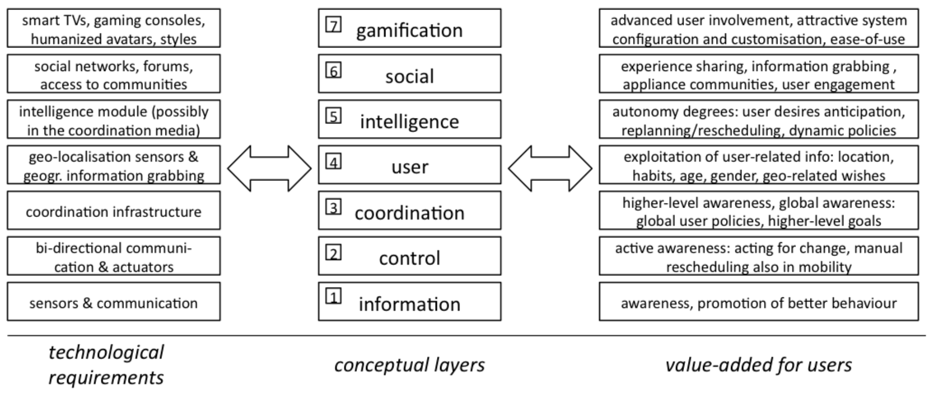
7.3.2 Butlers Model & Architecture

The Butlers architecture [Den14] defines a technology-neutral framework made of seven conceptual layers, relating technologies, features and the corresponding value-added for users (Figure 7.3). Although originally defined for the smart home context, it can be fruitfully specialised to STSS, KIE, and specifically to IoT scenarios.

The bottom layers concern the enabling technologies, such as mono or bi-directional communication-enabled sensors, meters, actuators, etc.; in the middle, infrastructural / middleware layers aim to provide coordination and geographical information services. The top layers focus on specific aspects, like intelligence, sociality, gamification: as such, they are not necessarily to be taken in the sequence. The resulting map can be used both to locate a system based on its feature or, conversely, to identify unexplored market niches.

Most of today's smart/domotic devices (e.g. [Ama16]), in particular, basically provide just remote monitoring or control facilities via some suitable Android/iOS app: so, they are conceptually located at level 2. The Butlers vision suggests that this is just the first chapter of the story: there's much more to be added to achieve real *Smart* Spaces. This is where the upper layers should come into play—and the conceptual map provides its key to what, how and why.

As discussed in [Den14] what is relevant here is that a smart space (e.g. a smart home) can interact with its users not only to monitor (level 1) or remote-control (level 2) the environment (e.g. the home appliances), but – via a suitable coordination infrastructure (level 3) – more generally to provide an *immersed, smart experience*, taking into account the users' habits, behaviour, location, preferences (level 4) to reason on the overall situa-



smart TVs, gaming consoles, humanized avatars, styles
social networks, forums, access to communities
intelligence module (possibly in the coordination media)
geo-localisation sensors & geogr. information grabbing
coordination infrastructure
bi-directional communi-cation & actuators
sensors & communication
7 gamification
6 social
5 intelligence
4 user
3 coordination
2 control
1 information
advanced user involvement, attractive system configuration and customisation, ease-of-use
experience sharing, information grabbing , appliance communities, user engagement
autonomy degrees: user desires anticipation, replanning/rescheduling, dynamic policies
exploitation of user-related info: location, habits, age, gender, geo-related wishes
higher-level awareness, global awareness: global user policies, higher-level goals
active awareness: acting for change, manual rescheduling also in mobility
awareness, promotion of better behaviour
technological requirements
conceptual layers
value-added for users

Figure 7.3: *Butlers multi-layer reference architecture, from [Den14]*

tion (level 5) and possibly anticipate the user's needs. In this view, social networks (level 6) can be further sources of user-related information, while gamification (level 7) can be essential for technology acceptance—a crucial success factor in STS, where the human factor is at least as relevant as advanced technicalogies.

Butlers for Smart Spaces (Figure 7.4) is the contextualisation of the Butlers vision to Socio-Technical Smart Spaces. In this context, some lower-level functionalities are typically provided by the underlying infrastructure, while some envisioned upper functionalities are too far from the foreseeable future or from the current state of the art, so their layers can be collapsed/dropped. This is why information (1) and control (2) layers are grouped together in a single *Monitoring* layer—the ability to act on the environment being a fundamental property of the Smart Space notion itself. Smart Spaces also grab a lot of raw information, which needs to be pre-processed to become exploitable knowledge: since this activity is somehow in-between information retrieval (layer 1) and coordination (layer 3), a single *Services* layer is introduced on top of the Monitoring layer.

Moreover, since users and environment are the main protagonists of a Smart Space, coordination must necessarily take users into account, so coordination (3) and user-aware (4) layers can also be conveniently grouped. Such coordination is likely to be complex enough to justify a clear separation between (general and user-specific) goals and policies, so that different policies can be developed for the same goals. Accordingly two mid-layers, *Goals* and *Policies*, are introduced side-by-side at that level, making a step towards pro-activity and situatedness. Moving up, the very nature of a Smart Space suggests that the reasoning about the surrounding environment, which shapes the "Space", can be conveniently separated from the "more basic" reasoning layer—for both conceptual and practical reasons. The *Reasoning* and *Situated reasoning* layers capture this separation, representing, respectively, the reasoning capabilities which exploit only the local/user knowledge, and which exploit also the surrounding environment. Gamification is left aside at this stage, as it is orthogonal to the tailoring of Butlers to the Smart Spaces context.

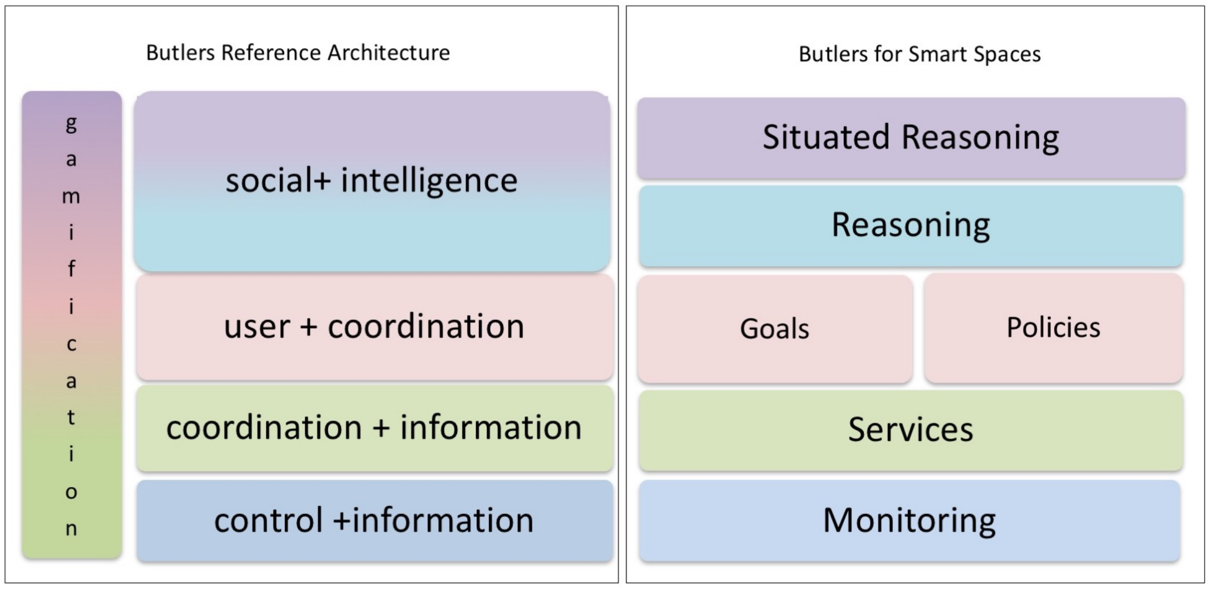


Figure 7.4: *Tailoring Butlers to the Smart Spaces context*

Chapter 8

The Home Manager

In this chapter we present *Home Manager* [Hom14], inspired to the Butlers for Smart Spaces architecture: a multi-paradigm, agent-based platform for the implementation of Smart Living contexts—particularly focused on the reasoning aspects, mainly to anticipate the users' needs. An overview of the current state of the Home Manager framework and technology is provided, that is, a prototype reflecting the extent to which the micro-intelligence model and approach has been actually implemented into a working system. For concreteness, running examples are presented.

The last section discusses describes how these running examples inspired the micro-intelligence approach discussed in this work, both from the point of view of the model and from the architecture.

8.1 The Home Manager

Home Manager [DC15, DCP14] is an open source platform [Hom14] for Smart Spaces, inspired to the above architecture and explicitly conceived to be open, deployable on a wide variety of devices (PCs, smartphones, tablets, up to Raspberry PI 2), and – thanks to the underlying TuCSoN [OZ99a] infrastructure and the suitable integration with tuProlog – suitable to accommodate "as much intelligence as the system needs, where the system needs".

Its purpose is to provide advanced services to users immersed in / interacting with the surrounding environment—in particular, the ability to reason on potentially any kind of relevant data, both extracted from the users's preferences and grabbed from other sources, so as to anticipate the users' needs.

Before discussing the platform, we shortly summarise the main features of TuCSoN infrastructure.

8.1.1 The TuCSoN infrastructure in a nutshell

TuCSoN [OZ99a, OZ99b] is a tuple-based agent coordination infrastructure for open distributed MAS, rooted on ReSpecT *tuple centres* [OD01a]. Moreover, two middleware abstractions play the role of boundary artefacts—namely, *Agent Coordination Contexts* (ACC) [Omi02] for agents, and *transducers* for resources.

Tuple centres are enhanced logic tuple spaces, distributed over a network of TuCSoN nodes, and programmable via the (Turing-equivalent) ReSpecT [Omi07] logic language: in fact, the ReSpecT virtual machine is itself built on top of tuProlog. In the TuCSoN vision, tuple centres embed the coordination laws, enabling MAS designers to govern the interaction space and reifying the "social intelligence". Moreover, ReSpecT support to *situatedness* [CO09, CO10] makes it possible to associate events in the interaction space – occurring as a consequence of agents activities or environment changes – to appropriate handlers (computations).

Each agent is also associated to an ACC, the security and organisation abstraction in charge of mediating the agent interactions with the MAS—in particular, providing the agent with available operations, based on its role and task. Transducers [CO09] represent individual resources, with their own peculiar ways of interacting: each transducer is capable of two-way interaction, to map meaningful resource events upon admissible MAS events.

The TuCSoN technology is light-weight, open source [tuc08], and Java-based—although some (limited) interaction with Windows 10-IoT is also possible [Mar16b].

8.1.2 Butlers on Home Manager

Figure 8.1 shows how the five layers of Butlers for Smart Spaces re-shape on the Home Manager platform. The TuCSoN infrastructure surrounds and encompasses all layers, as it enables the seamless integration of heterogeneous entities, bridges among technologies and agents' perceptions, and supports situated intelligence.

Each device is equipped with an agent, which acts as a sort of "proxy" to bring the physical device into the agent society that powers the Smart Space. At a basic stage, this agent enables the device monitoring and (possibly) remote control, grabbing the necessary information through TuCSoN sensors and probes, and acting on the environment via its actuators and transducers.

The Services layer takes the concrete form of *Service-Level Agents* and *Basic Policies*: the idea is that agents in this layer perform some information elaboration and possibly retrieval via mechanisms that, however, do not require sophisticated reasonings—for instance, grabbing information from weather web sites, or from selected Twitter pages, based on the selected basic policies, such as the user's preferred weather sites or followed Twitter people.

Analogously, Goals and Policies take the concrete form of *Goal-Oriented Agents* and

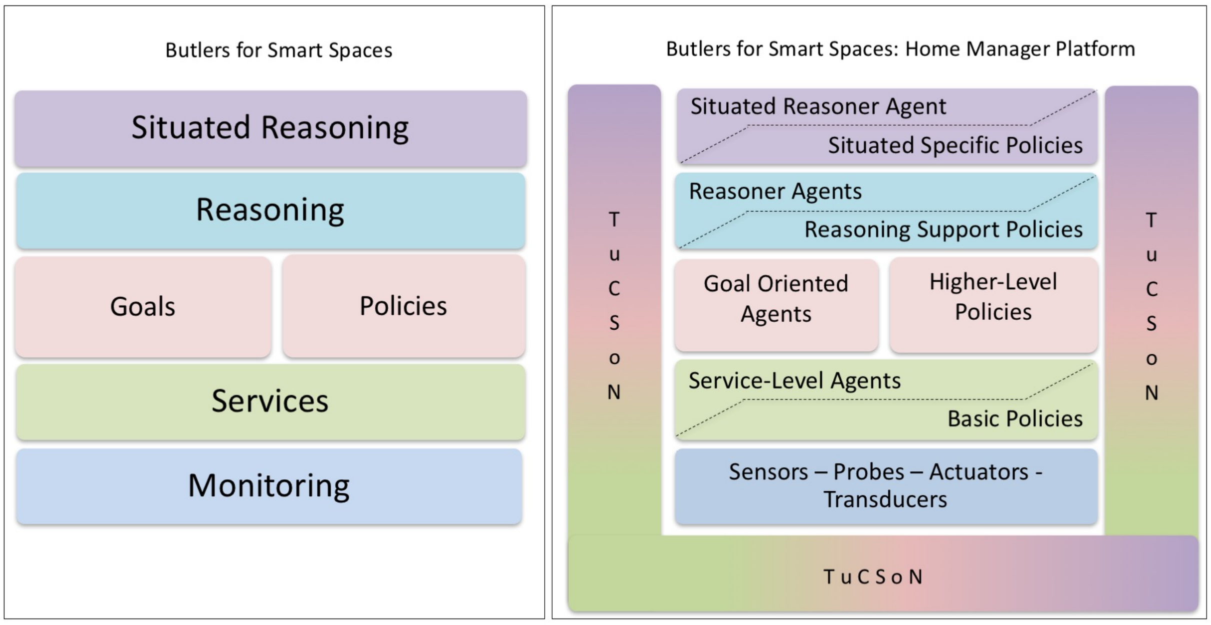


Figure 8.1: *Butlers for Smart Spaces on Home Manager*

Higher-Level policies, respectively. Policies at this stage typically concern everyday life habits and aspects – such as children not being allowed to set any twitter policy, etc.; so, they are generally rather stable. Accordingly, the Goal-oriented Agents are charged of autonomous decisions based on such policies: for instance, in the Twitter service case, the agent goal could be to retrieve suitable tweets from selected people and highlight the ones that, say, receive more than 100 likes, or refer to given topics, etc.

More complex, intelligent and fine-tuned behaviours call for further reasoning: *Reasoner Agents* are charged of potentially any kind of reasoning over user-related knowledge – typically the user's profile, habits, and preferences –, while *Reasoning Support Policies* encapsulate the corresponding rules. The top layer extends such capabilities towards situatedness, in time and space: *Situated Reasoner Agents* take into account the user location, movement, etc. to make situated deductions and perform real-time suggestions and pro-active actions: e.g., in the Twitter case, a reasoner could decide to include further Twitter pages if the user is moving to another city, assuming she might desire to receive travel/destination information (weather forecast, traffic, entertainments, etc.). *Situated Specific Policies* encapsulate the corresponding rules.

In Home Manager, *Butlers for Smart Spaces* layers are concretised into a TuCSoN-based MAS. This is why the TuCSoN infrastructure surrounds all layers, enabling the seamless integration of heterogeneous entities, bridging among technologies and agents' perceptions, and supporting situated intelligence.

From the architectural viewpoint, each device is supposed to be equipped with an agent, acting as a sort of "proxy" to interface the physical device to the agent society

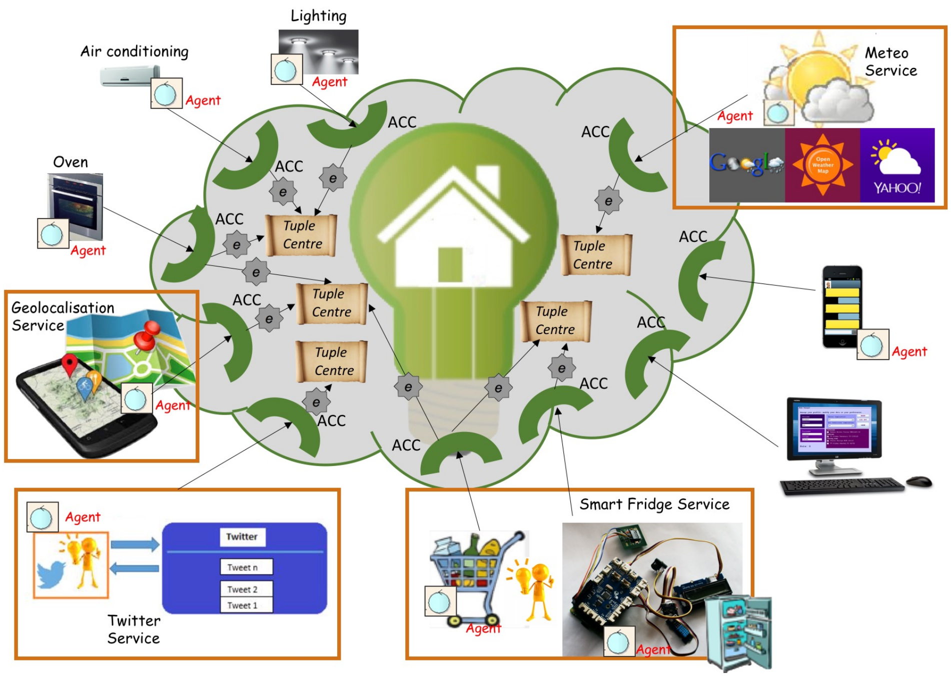


Figure 8.2: *The Home Manager logical architecture*

that powers the Smart Space. Then:

- in the *Monitoring* layer, proxy agents provide for the device monitoring and (possibly) remote control, interacting via TuCSoN to grab the required information and operate on the environment;
- the *Services* layer is concretely split into *Service-Level Agents* and *Basic Policies*, following the idea that information is elaborated in this layer via mechanisms that do not require sophisticated reasonings (e.g. grabbing weather info based on the simple user policies, like the preferred weather sites);
- analogously, *Goals* and *Policies* take the concrete form of *Goal-Oriented Agents* and *Higher-Level policies*: at this stage, policies concern everyday life habits and aspects, and are generally rather stable; the corresponding agents handle the autonomous decisions which refer to such policies;
- the *Reasoning* layer also splits into *Reasoner Agents* and related *Reasoning Support Policies*, reasoning on user-related knowledge (profile, habits, preferences) and corresponding rules;
- similarly, the *Situated Reasoning* layer concretises into *Situated Reasoner Agents* (which take into account the user location, movement, etc. to provide real-time suggestions and pro-active actions) and related policies.

8.1.3 The technology

The Home Manager platform is built on top of three main technologies: TuCSoN and its artifacts – tuple centres and ACCs – for MAS coordination, the ReSpecT logic language to bring situated intelligence to TuCSoN nodes, and tuProlog to build light-weight intelligent agents.

The logical architecture is shown in Figure 8.2: each device is assumed to be equipped with an agent, which is connected to a TuCSoN ACC defining its admissible operations and roles in the agent society; as such, they embed the individual intelligence. Tuple centres, on the other hand, embed the "social intelligence", further supported by tuple centres' *linkability*—the ability to trigger reactions in other tuple centres as a consequence of a local event [ORZ06].

The infrastructure embeds and enforces the coordination laws to mediate among agents, governing the agent-agent and the agent-environment interaction. Heterogeneous entities, such as legacy agents, can be integrated by charging the infrastructure of bridging the gap between the common ontology and the specific agents' representations and ontologies [OD01b]: agents can thus be heterogeneous in nature, implementation language, etc.—the only requirement being that they coordinate via the TuCSoN APIs, adhering to their intended semantics.

The prototype enables users to control the system configuration and interaction, yet with no need to know or operate directly on the underlying machinery—i.e., the inner tuple-based representation of data and policies.

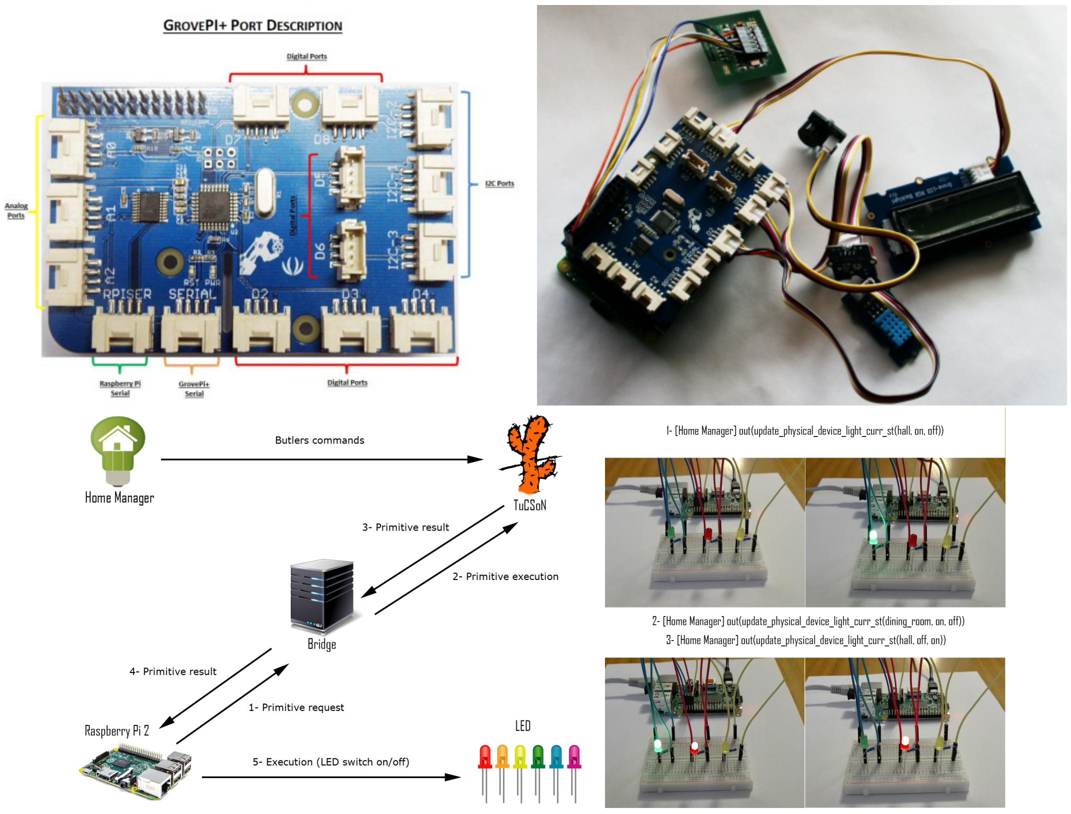
It is worth highlighting that the declarative, tuple-based approach is what makes it easy to evolve the system incrementally from a purely-simulated environment hosted on a personal computer (with simulated house, inhabitants, and sensors) to an "increasingly-real" system, interfaced – for instance – to an Android smartphone as shown in the following sections, so that users can interact with the system in mobility via a suitable app—which, by the way, takes care of geo-localising the user and to support advanced services based on the user's situatedness in space and time (Subsection 8.3).

Going farther, the system can also be interfaced to handling actual hardware devices, up to possibly run "out of the box" on low-cost stand-alone platforms like a Raspberry PI 2 Model B. The latter choice provides *i)* an independent installation on a dedicated software+hardware platform, dropping the requirement of a personal computer for the hosting environment, and *ii)*, perhaps more relevantly, the chance to exploit the many Raspberry sensors and devices.

8.1.4 The Raspberry porting

For our intended application context (Subsection 8.2.1), the support of displays and RFID readers is particularly relevant, to simulate the presence and movement of items, users, etc.; but several other (low cost) sensors and actuators are necessary for a reasonable simulation. This is why we developed our prototype on top of the GrovePi [Gro] board, which comes with many nice sensors and actuators (displays, switches, temperature sensors, etc.) in an all-in-one pack with customized Raspbian version: the resulting platform [Car15] is shown in Figure 8.3 (top).

Figure 8.3: *Home Manager out-of-the-box on a Raspberry Pi 2 + GrovePi kit (top); interaction with a Windows-10 system – architecture and prototype (bottom).*


GrovePI+ Port Description
Digital Ports
Analog Ports
I2C Ports
Raspberry Pi Serial
GrovePi+ Serial
Digital Ports
Butlers commands
Home Manager
TuCSoN
3- Primitive result
2- Primitive execution
Bridge
4- Primitive result
1- Primitive request
Raspberry Pi 2
5- Execution (LED switch on/off)
LED
1- [Home Manager] out(update_physical_device_light_curr_st(hall, on, off))
2- [Home Manager] out(update_physical_device_light_curr_st(dining_room, on, off))
3- [Home Manager] out(update_physical_device_light_curr_st(hall, off, on))

Even more interestingly, and orthogonally, the Raspberry can be exploited as an implementation platform for smart devices – Smart Fridge, Smart Oven, etc. Although Java and a Raspbian-based Raspberry is the most obvious choice, a multi-platform, multi-language environment could be another, challenging, perspective. To this end, we explored the Microsoft Windows 10-IoT Core [Mic15a, Mic15b] platform, which enables UWP (Universal Windows Platform) applications to be designed in Visual Studio and then deployed to the Raspberry PI, supporting remote executing and debugging.

In order to integrate it into Home Manager, an ad-hoc bridge has to be set up to interface the (Java-based) TuCSoN primitives, used by Home Manager, with the Windows 10 platform (Figure 8.3, bottom left): to ensure that the communication with TuCSoN is seamless, the bridge itself is written in Java. The (C#-coded) client agent and the bridge communicate via UTF-8 strings: a suitable XML configuration file specifies the data required by the client agent (bridge IP and port numbers, target tuple centre name, etc.) and maps the client coordination language primitives onto the TuCSoN one. Figure 8.3 (bottom right) shows a trivial demo application, where some LED are controlled via such a bridge.

8.2 Building Smart Spaces on Home Manager

According to Figure 8.1, and following the logical architecture in Figure 8.2, building a smart space on top of Home Manager means *i)* to identify the device and service categories that are relevant for that specific space; *ii)* to define a suitable tuple-based representation of the relevant knowledge, and *iii)* the desired agent interaction protocols – which, by the way, need not match the knowledge representation 1-1, since tuple centres can be suitably programmed to bridge the gap; *iv)* to develop an agent for each device category and for each external service to interact with – possibly testing and debugging them separately from the rest of the system, thanks to the data-driven approach. The whole design process is aimed at keeping the social/individual intelligence, on the one hand, and mechanisms/policies, on the other, clearly separate.

8.2.1 Application scenario

The intended application scenario is a smart house immersed in a smart living context, with devices (air conditioners, lights, etc.) and users of different categories and (RBAC-based) roles [DCP14].

At the basic operation level, the goal is to satisfy the users' desires (e.g. room light, temperature) while respecting some global constraints (e.g. energy saving, temperature range, etc.): as an example, Table 8.1 shows the ReSpecT reactions that control the temperature of a room, averaging the preferred temperatures of users in case two or more people are present.

At a higher level, however, the goal is more ambitious—to anticipate the user's needs by reasoning on the user's habits and on any user-related information, including the environment where he lives, travels, purchases goods, etc. The idea is to go beyond the mere monitoring and remote control of house appliances via app, as it is often found today, towards:

- exploiting the user's location, tracked by the smartphone GPS, to enable an intelligent reasoner agent to take autonomous "situated" decisions;
- explore the environment around the user's location, extracting information about shops, services, etc, to be taken as a further reasoning knowledge base;
- getting information about the surrounding environment (e.g. weather) so as to tailor decisions to the user's habits and needs;

```
% 2+ Users
reaction(
    in(new_temperature(L,Users,T)),
    (request),
    ( Users > 1,
        out(avg_temp(L,L,Users,0))
    )
).
% Calculate the temperature..
reaction(
    out(avg_temp(L,[user_pref(X,WarmTemp,W,Z)|OtherPrefs],Users,Sum)),
    (internal),
    ( in(avg_temp(L,[user_pref(X,WarmTemp,W,Z)|OtherPrefs],Users,Sum)),
        rd(temp_mode(heat)),
        NewSum is Sum+WarmTemp,
        out(avg_temp(L,OtherPrefs,Users,NewSum))
    )
).
reaction(
    out(avg_temp(L,[user_pref(X,Y,CoolTemp,Z)|OtherPrefs],Users,Sum)),
    (internal),
    ( in(avg_temp(L,[user_pref(X,Y,CoolTemp,Z)|OtherPrefs],Users,Sum)),
        no(temp_mode(heat)),
        NewSum is Sum+CoolTemp,
        out(avg_temp(L,OtherPrefs,Users,NewSum))
    )
).
reaction(
    out(avg_temp(L,[],Users,Sum)),
    (internal),
    ( in(avg_temp(L,[],Users,Sum)),
        Average is Sum/Users,
        out(new_temperature(L,Users,Average))
    )
).
```

Table 8.1: *Excerpt of* ReSpecT *code for temperature management from [CD16a]*

- interacting with selected social networks (e.g. Twitter) to grab information that could later be exploited fur further reasonings;
- tracking the human presence for intrusion detection or elderly applications (e.g. to detect falls, stand or walk status, etc.);
- overall, providing novel, integrated services by coupling smart appliances (smart fridge, smart ove, etc.) with environment and user information.

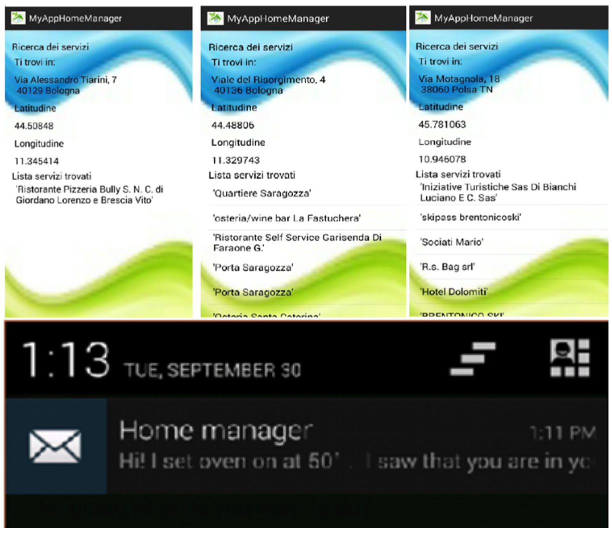
8.3 Space situatedness via geo-localisation

As a first step in supporting the user's situatedness in the environment in time and space, [DC15] presented a simple scenario exploiting the geo-localisation facility embedded in any modern smartphone to:

- extend the system intelligence, recognising places and services based on their position, via the Google Places API;
- provide user-location-related information e.g. about surrounding services.

There, the user location is monitored and conceptually used to control a Smart Oven, switching it on automatically if the user buys a take-away pizza in her way back home: Figure 8.4 shows the exploration of the surrounding services in the Android app at different user's positions [Pao15]. As a further consequence, the smart house switches on the oven,

Figure 8.4: *Exploration of surrounding services, and notification for autonomous actions*



MyAppHomeManager

Ricerca dei servizi
Ti trovi in:
Via Alessandro Tiarini, 7
40129 Bologna
Latitudine
44.50848
Longitudine
11.345414
Lista servizi trovati
'Ristorante Pizzeria Bully S. N. C. di Giordano Lorenzo e Brescia Vito'

MyAppHomeManager

Ricerca dei servizi
Ti trovi in:
Viale del Risorgimento, 4
40136 Bologna
Latitudine
44.48806
Longitudine
11.329743
Lista servizi trovati
'Quartiere Saragozza'
'osteria/wine bar La Fastuchera'
'Ristorante Self Service Garisenda Di Faraone G.'
'Porta Saragozza'
'Porta Saragozza'

MyAppHomeManager

Ricerca dei servizi
Ti trovi in:
Via Motagnola, 18
38060 Polsa TN
Latitudine
45.781063
Longitudine
10.946078
Lista servizi trovati
'Iniziative Turistiche Sas Di Bianchi Luciano E C. Sas'
'skipass brentonicoski'
'Sociati Mario'
'R.s. Bag srl'
'Hotel Dolomiti'

1:13 TUE, SEPTEMBER 30

Home manager 1:11 PM
Hi! I set oven on at 50° . I saw that you are in yo

so that the user find it hot enough for warming the pizza as she comes back. The system decision is notified to the user (Figure 8.4, right), so that she always remains in control and has the last word.

A possible evolution, envisioned in [Den14], could be to exploit such info not to overcome the electrical power threshold, e.g. postponing the washing machine to switch on the oven without causing a blackout—which would be particularly dangerous after dusk, in presence of elderly people or children, etc.

8.4 Space and time situatedness based on weather

Weather info is by nature situated in time and space, and – even more relevant – suitable to inherently and continuously condition everyone's life: many "micro-decisions" we take every day depends on weather—doing laundry, to (not) go out for shopping, switching on heating, closing windows, etc.

This is why a Home Manager agent has been developed for that purpose (Figure 8.5) [Cel15]: once retrieved, weather info can be later exploited in several ways—from intelligent appliances scheduling (e.g., avoid scheduling the washing machine in a raining day, so as to avoid the dryer), to the automatic control of rolling shutters based on sunrise and sunset times, to just taking into account the user's mood, etc. For instance, intelligent shutter control could be obtained by embedding a small piece of intelligence – a ReSpecT reaction – in the infrastructure, so as to intercept the addition of new relevant weather data (independently of the specific tuple format) and generate the shutter actions.

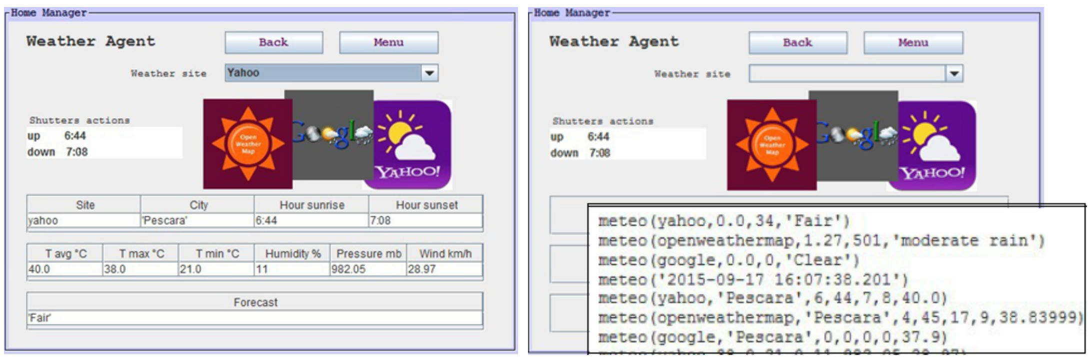
Weather sources can be Google's [Goo14b], Yahoo's [Yah14] or OpenWeatherMaps [Ope14]'s web services—but they all require the house location. This datum can either be statically stored in some tuple centre or, better, be dynamically extracted from the house's IP number, thanks to services like `IP-API.com`, and then turned onto a physical city name, via Flicker or similar services. The retrieved info is reified in form of suitable tuples (Figure 8.5, bottom right) in the `weather-tc` tuple centre: the history is maintained, so as to enable any potential reasoning.

8.5 Let's be social: Twitter integration

As recalled in Figure 8.1, the social aspects have been included in the Butlers framework from its very origin—mainly as a source of further knowledge about the user (habits, interests, traffic, special offers, government warnings, etc.), but also, more in the long-term perspective, of butlers networks [Den14].

The first choice has been to develop a Twitter agent (Figure 8.6) [Bev15] – both because Twitter's text-based nature makes it easier to parse and extract data, and because messages are typically more informative than, say, Facebook posts, which are more often oriented to closed groups of friends commenting on their own lives.

Figure 8.5: *Weather agent and the respective tuples*



```
up      6:44
down    7:08
```

Site	City	Hour sunrise	Hour sunset
yahoo	'Pescara'	6:44	7:08

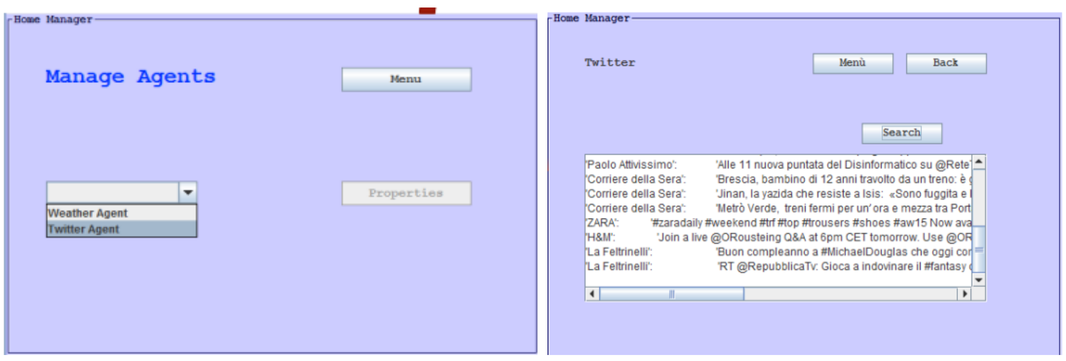
T avg °C	T max °C	T min °C	Humidity %	Pressure mb	Wind km/h
40.0	38.0	21.0	11	982.05	28.97

Forecast
'Fair'

```
meteo(yahoo,0.0,34,'Fair')
meteo(openweathermap,1.27,501,'moderate rain')
meteo(google,0.0,0,'Clear')
meteo('2015-09-17 16:07:38.201')
meteo(yahoo,'Pescara',6,44,7,8,40.0)
meteo(openweathermap,'Pescara',4,45,17,9,38.83999)
meteo(google,'Pescara',0,0,0,0,37.9)
```

To keep things simple, this early prototype considers a single Twitter user, representing the home butler, with read-only capabilities: the goal is to monitor selected "interesting" users, and – like the weather info above – store the related tweets in a suitable tuple centre, for further reasoning (Figure 8.6, bottom right). Technically, Twitter is accessed

Figure 8.6: *The Twitter agent*



```
'Paolo Attivissimo':      'Alle 11 nuova puntata del Disinformatico su @Rete
'Corriere della Sera':    'Brescia, bambino di 12 anni travolto da un treno: è
'Corriere della Sera':    'Jinan, la yazida che resiste a Isis:  «Sono fuggita e
'Corriere della Sera':    'Metrò Verde,  treni fermi per un'ora e mezza tra Port
'ZARA':       '#zaradaily #weekend #trf #top #trousers #shoes #aw15 Now ava
'H&M':        'Join a live @ORousteing Q&A at 6pm CET tomorrow. Use @OR
'La Feltrinelli':         'Buon compleanno a #MichaelDouglas che oggi co
'La Feltrinelli':         'RT @RepubblicaTv: Gioca a indovinare il #fantasy
```

via REST API, with OAUth [OAu] to handle the user authentication and Twitter4J [Twi] for Java inter-operability. Due to its nature of early implementation, severe limitations apply—in particular, Twitter credentials are currently stored in a tuple centre; security issues are to be engineered in a future release.

In the longer-term perspective, however, the butler account could also post messages—e.g. to share result of its reasonings with its own followers.

8.6 A Context Reasoning case study: The Smart Kitchen

In this Section, we discuss the case of the Smart Kitchen, made of a *Smart Fridge*, a *Smart Pantry*, a *Smart Oven*, a *Smart Mixer*, integrated with a *Smart Shopper* butler

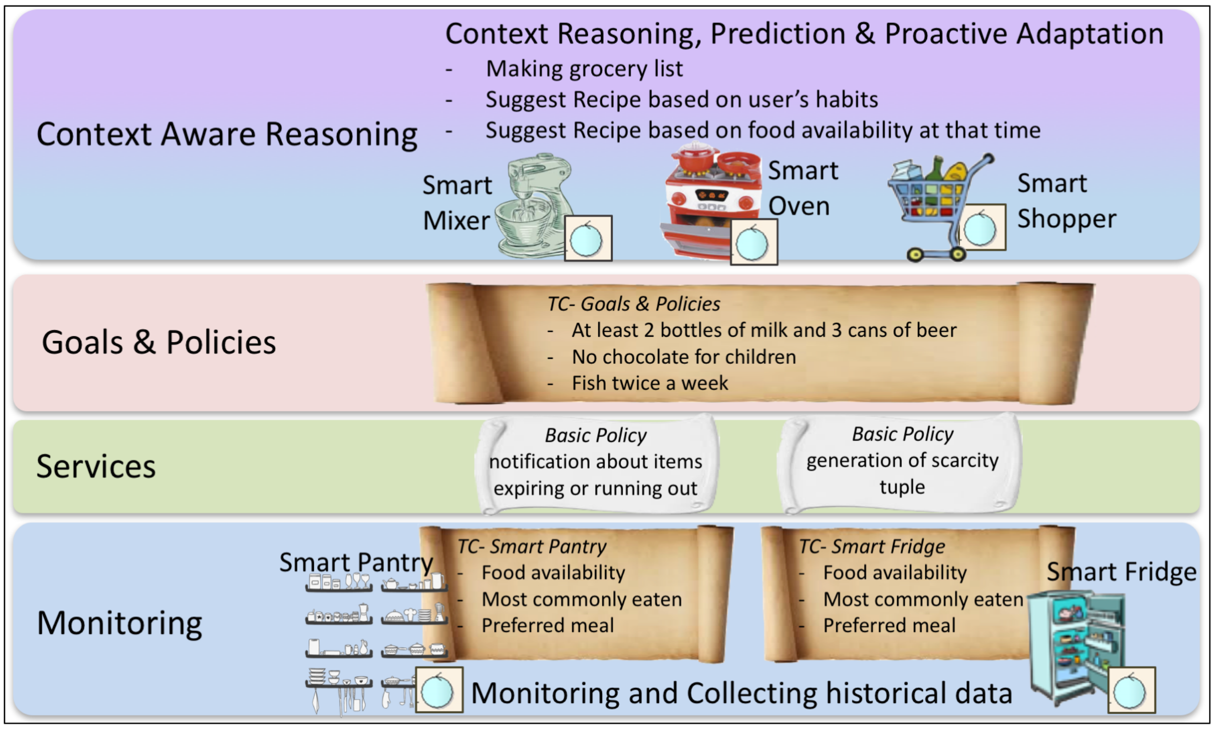


Figure 8.7: *The Smart Kitchen Scenario*

aimed at managing the food supply.

Figure 8.7 shows the envisioned scenario, highlighting the layers described in the previous section. Bottom-up, the Smart Fridge and the Smart Pantry are capable of monitoring the quantity of food, and of collecting historical data on user's habits, e.g. the most commonly eaten food and preferred meals. All data are reified on selected tuple centres to create a knowledge base, usable by upper-level reasoning agents (like the Smart Shopper) to predict the user's needs or make contextualised suggestions. In the Services level, whenever a product is taken from the fridge, suitable Basic Policies check its quantity against user-defined, per-product thresholds, and generate the corresponding `buy` tuple if necessary (details in [CD16a]). No Service-Level agents are devised at this layer.

Goals and Policies refer to user preferences and constraints, like making sure that there are at least 2 bottles of milk, that fish is cooked twice a week, etc.

At the upper layer, the Smart Shopper butler (Figure 8.8) compiles the shopping list based on the above tuples, contacting the "proper" vendor via the "appropriate" means: what "proper" and "appropriate" stand for depends on context-aware policies. Vendor selection could be based e.g. on fidelity cards, promotional campaigns, distance, consumer habits, etc., while contact means could be email, online shopping, up to "ask your neighbour", etc. Policies could also require that the shopping cart total reaches a minimum amount to get free home delivery, that multiple markets are compared to find

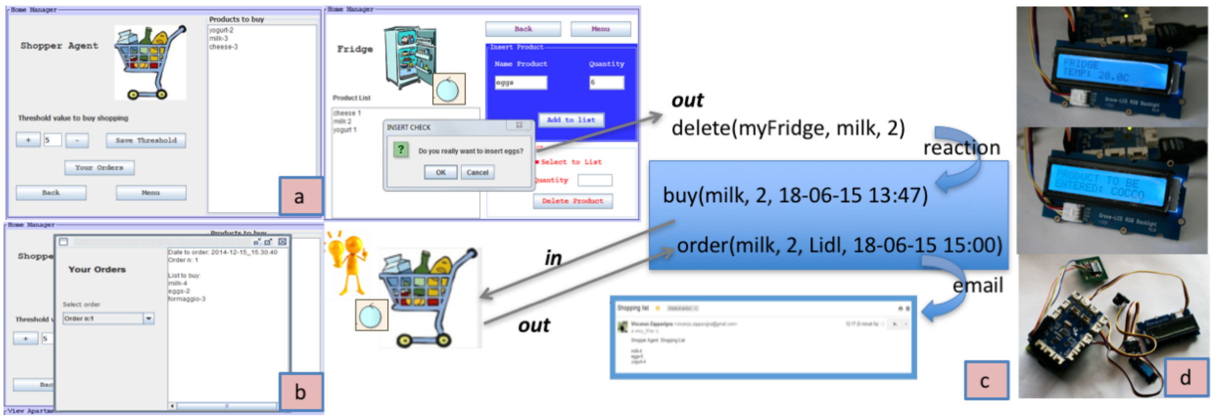


Figure 8.8: *Policies setup (a), order monitoring (b), Shopper Agent (c), Hardware (d)*

the most convenient—possibly taking into account fidelity cards and special offers; and so on.

In its turn, the Smart Oven aims to support the user's food cooking—in principle, exploiting any available technology to identify and cook the food; the user profile is supposed to include information about his/her dietary requirements. The Smart Mixer manages the recipe instructions, interacting with both the Smart Fridge – to check that the ingredients for the selected recipe are actually available – and with the Smart Oven – to check its ability to cook that food and potentially synthesise the proper control instructions. Since recipes are prepared based on the current content of Smart Pantry and Smart Fridge, the Smart Mixer behaviour aims to exhibit a (primitive) form of context adaptation.

In the current experimental prototype (Figure 8.8), the Smart Oven, Mixer and Pantry are simulated in software, while the Smart Fridge integrates software-only and software+Raspberry+GrovePI hardware (for display, sensors and LEDs), plus an RFID tag reader for tracking the content [CD16a].

Given the proof-of-concept nature of this prototype, policies are intentionally kept simple. The Smart Fridge monitors the fridge content, reified as `fridge_content/4` tuples: a product is *scarce* when its quantity falls below a pre-defined threshold, expressed as another suitable tuple. So, whenever an item is taken out of the fridge, the policy performs the above check and generates a `scarcity/5` tuple if this is the case. Based on this information, other policies generate the purchase orders, in the form of `buy/3` tuples: the current (trivial) policy is to produce an order when "enough" (user-defineable) `scarcity` tuples have been accumulated. The result is a suitable `buy(`*`product,quantity,timestamp`*`)` The Smart Shopper finally consumes these tuples and, based on its own policies, decides the quantities to be purchased, and sends the order to the "proper" (currently: the pre-defined) vendor via the "appropriate" means (currently: by email).

The middleware infrastructure encapsulates the coordination laws, enabling interoperability and integration with third-part services, like the mailing service that sends shopping orders. Apart from the knowledge base, the declarative approach provides a *lingua franca* to bridge among the different forms of heterogeneity, supports the agent uncoupling and the separation between policies and mechanisms, and supports context reasoning.

Of course, it is up to the designer to balance between the intelligence to be embedded in higher-level policies and to be put onto reasoning agents: generally speaking, agents can be expected to focus on opportunistic behaviour, while policies on synthesising the information for higher-level reasonings—especially to take into account environmental time- and space-situatedness. For instance, opportunistic behaviour could exploit the user's location to alert of a nearby market, minimising the time spent in traffic (and therefore also fuel consumption and cost), or suggest an alternative market to avoid traffic jams; and so on.

8.7 Discussion & Remarks

As highlighted by the above running examples, enabling the ambient intelligence IoT vision means that consumers will be provided with universal and immediate access to available content and services, together with ways of effectively exploiting them. From a software engineering standpoint, this means that the actual implementation of intelligent behaviours requested by a user can only be resolved at runtime according to the user's specific situation. In particular, all the proposed examples emphasise the requirement in ambient intelligence, and more generally in pervasive computing, of not solely ubiquitous computing (i.e., useful, pleasant and unobtrusive presence of computing devices everywhere) but also of ubiquitous networking (i.e., access to network and computing facilities everywhere) and of *intelligent aware interfaces* (i.e., perception of the system as intelligent by people who naturally interact with the system that automatically adapts to their preference).

While available technologies provide us with base enablers of the ambient intelligence vision, there is still a long way to go before offering robust ambient intelligence systems to consumers, requiring advances in most areas relating to the computer science field: e.g., hardware supporting low-power high-performance wireless devices, network ensuring connectivity everywhere, human-computer interaction enabling intelligent multi-modal interfaces, development support for deploying ubiquitous applications.

To this end, the micro-intelligence vision (chapter 1 Section 1.1) supports the abstract specification of intelligence together with its dynamic composition according to the *environment.* The proposed model builds on the SOA architecture, whose pervasiveness enables both services availability in most environments, and specification of applications supporting automated retrieval and composition. Moreover, as a key aspect for IoT pervasive scenarios, it defines a *unique standard interface.*

The LP micro-intelligence approach addresses support for distributing intelligence in pervasive scenarios, concentrating more specifically on enabling seamless access to content and services any-time, any-where. The key feature of our approach consists of enabling the dynamic composition of the intelligence resource, possibly distributed, offering it as services according to the mobile users' situation.

Reinterpreting the LP as a service and enhancing the model with the label concept lay the basis for context aware reasoning, proactive recommendation and situations prediction, in that enable a standard access to *situated intelligence*. The composable nature of SOA enable dynamic (runtime) composition and exploitation of intelligent behaviours, while the heterogeneous nature make it possible to run on a variety of platforms and small object and constrained resourced hardware. Finally, the SOA vision guarantees actors agree upon standard exchange protocols and languages to interact with each other (e.g., XML, WSDL, SOAP, UDDI).

The Home Manager case study provides both an inspiration for the micro-intelligence model –highlighting requirements and constraints of IoTreal scenarios– and a testbed system to verify the model choice.

With respect to existing ambient intelligence framework, its main originality is that it enables IoT systems to face the distribution of intelligence (from the design phase) as a stand-alone aspect, without loss of integration with all other components. Moreover, it shows how relevant emergent behaviours of IoT can be deal by focussing on local situated intelligence, in that pervasive systems promote designing and developing applications in terms of autonomous software entities, situated in an environment, and that can flexibly achieve their goals by interacting with one another in terms of high-level protocols and languages.

Of course, many advances have to be done in order to deeper understand the advantages and limitations of the framework in terms of performance and responsiveness. Performance measurement seeks to monitor, evaluate and communicate the extent to which various aspects of the health system meet key objectives.

In particular, the provision of relevant, accurate and timely performance information is essential for assuring and improving the performance of pervasive systems. Citizens, patients, governments, politicians, policy-makers, managers and clinicians all need such information in order to assess whether smart systems are operating as well as they should and to identify where there is scope for improvement. Without performance information, there is no evidence with which to design health system reforms; no means of identifying good and bad practice; no protection for patients or payers; and, ultimately, no case for investing in the health system.

Part IV

Conclusion & Future Work

Chapter 9

Conclusion & Future Work

This thesis deals with micro-intelligence issues in IoT systems, proposing a new approach based on LPaaS and LVLP.

The basic idea is to spread small inference engines over the distributed system, aimed at providing local, situated reasoning capabilities to contribute to the global system goals. Accordingly, the proposed framework extends LP to face the challenges of today pervasive systems, by providing the models and technologies required to effectively support distributed situated intelligence, while preserving the features of declarative programming. The resulting architectural and linguistic approach relies on an inference engine with deduction capabilities available in term of services and extensible through suitably-tailored labelled models. We also present, discuss and evaluate the `tu`Prolog model and technology for micro-intelligence in IoT systems.

The rationale is to engineer effective intelligent mechanism for large-scale, data-intensive systems with “humans in the loop” —notably a difficult task, to be approached in a holistic way by considering the model, the architecture, and the domain specificity of each node. The reinterpretation of LP mechanisms in terms of service architecture and extensibility to domain-specific situation is what makes them capable of dealing with distribution, decentralisation, unpredictability, and scale in a simple yet expressive way. The service-based approach, in particular, *(i)* promotes the representation and reasoning with situations using a declarative language, providing a high level of abstraction; *(ii)* supports the incremental construction of context-aware systems by providing modularity and separation of concerns; *(iii)* promotes the cooperation and interoperation among the different entities of a pervasive system; and *(iv)* enables reasoning over data streams, like those collected by sensors.

Accordingly, the main contributions of this thesis can be summarised as follows:

- the definition of LPaaS approach for *distributed situated intelligence* as the natural evolution of LP in nowadays pervasive computing systems;
- the definition of a novel theoretical framework, LVLP, where different domain-

specific computational models can be expressed via labelled variables, capturing suitably-tailored labelled models;

- a novel software framework providing not only the core functionalities of the aforementioned model, but also those the satellite services necessary for spreading intelligence in a real-world pervasive system deployment.

Open issues to be addressed in the future research include:

- a specialised LP-oriented middleware, dealing with heterogeneity of platforms as well as with distribution, life-cycle, interoperability, and coordination of multiple situated Prolog engines – possibly based on the existing `tu`Prolog technology and TuCSoN middleware [OZ99a] – so as to explore the full potential of logic-based technologies in IoT scenarios and applications;
- the extension of the LPaaS interface with specific *space awareness* methods other than the existing time-aware methods: for instance, a `solveNeighbours` primitive could be added to consider the space around the client or the server, exploring the chance to opportunistically federate LP engines by need as a form of dynamic service composition;
- deeper exploration and better understanding of the consequences of applying labels to formulas, as suggested by Gabbay [Gab96];
- the application of the LVLP framework to different scenarios and approaches—such as probabilistic LP [SAFP15], the many CLP approaches [Coh90], distributed ASP reasoning [DP09], and action languages [DFP13]
- further formal investigation of uniform primitives expressiveness, for better framing the relative expressiveness of probabilistic coordination languages;
- the improvement of `tu`Prolog for IoT prototype (already under development).

Performance and responsiveness are other key issues in pervasive and service-oriented scenarios, especially considering the huge amounts of data possibly generated by IoT devices: so, in the perspective, we also mean to address such issues more in depth.

Finally, it must be highlighted that the case studies are currently implemented as proofs of concept, to demonstrate the model properties and the feasibility and effectiveness of the proposed approach, but do not constitute a full implementation usable e.g. for actual test performance. For such a reason, in the perspective we plan to implement the system in a real pervasive scenario, that enables performance measurement both in comparison to other non-logic based systems or possibly different logic based systems.

Bibliography

[ADB$^+$99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, UK, 1999. Springer-Verlag.

[AGR02] Alberto Artosi, Guido Governatori, and Antonino Rotolo. Labelled tableaux for nonmonotonic reasoning: Cumulative consequence relations. *Journal of Logic and Computation*, 12(6):1027–1060, December 2002.

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

[AIMN12] Luigi Atzori, Antonio Iera, Giacomo Morabito, and Michele Nitti. The social internet of things (siot) - when social networks meet the internet of things: Concept, architecture and network characterization. *Comput. Netw.*, 56(16):3594–3608, November 2012.

[AK15] C. Alexakos and A. P. Kalogeras. Internet of things integration to a multi agent system based manufacturing environment. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, Sept 2015.

[Ama16] Amazon Smart Home. https://www.amazon.com/smarthome-home-automation/b?ie=UTF8&node=6563140011, 2016.

[AMSM08] Shorr AF, Zilberberg MD, Micek ST, and Kollef MH. Prediction of infection due to antibiotic-resistant bacteria by select risk factors for health care–associated pneumonia. *Archives of Internal Medicine*, 168(20):2205–2210, 2008.

[AN04] Juan C. Augusto and Chris D. Nugent. The use of temporal reasoning and management of complex events in smart homes. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, pages 778–782, Amsterdam, The Netherlands, The Netherlands, 2004. IOS Press.

[AN06] Juan Carlos Augusto and Chris D. Nugent. *Designing Smart Homes: The Role of Artificial Intelligence (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[App14] Apple Home Kit. https://developer.apple.com/homekit/, 2014.

[ASF+14] Artur Arsénio, Hugo Serra, Rui Francisco, Fernando Nabais, João Andrade, and Eduardo Serrano. Internet of Intelligent Things: Bringing artificial intelligence into things and communication networks. In *Inter-cooperative Collective Intelligence: Techniques and Applications*, volume 495 of *Studies in Computational Intelligence*, pages 1–37. Springer Berlin Heidelberg, 2014.

[Aug05] Juan Carlos Augusto. Temporal reasoning for decision support in medicine. *Artif. Intell. Med.*, 33(1):1–24, January 2005.

[BA10] A. Bikakis and G. Antoniou. Defeasible contextual reasoning with arguments in ambient intelligence. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1492–1506, Nov 2010.

[BB94] Silvana Badaloni and Marina Berati. *Dealing with time granularity in a temporal planning system*, pages 101–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[BB06] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *Intelligent Systems, IEEE*, 21(2):10–19, 2006.

[BBH+10] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180, 2010. Context Modelling, Reasoning and Management.

[BC91] Antonio Brogi and Paolo Ciancarini. The concurrent language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, January 1991.

[BCG07] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology).* John Wiley & Sons, 2007.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, New York, NY, USA, 2003.

[BCR09] Oliver Brdiczka, James L. Crowley, and Patrick Reignier. Learning situation models in a smart home. *Trans. Sys. Man Cyber. Part B*, 39(1):56–63, February 2009.

[BEG+18] Gerhard Brewka, Stefan Ellmauthaler, Ricardo Gonçalves, Matthias Knorr, João Leite, and Jörg Pührer. Reactive multi-context systems: Heterogeneous reasoning in dynamic environments. *Artificial Intelligence*, 256:68 – 104, 2018.

[Ben86] Jon Bentley. Programming pearls: Little languages. *Commun. ACM*, 29(8):711–721, August 1986.

[Bev15] Sara Bevilacqua. Home intelligence & social network in the butlers perspective. Bachelor's Thesis – Scuola di Ingegneria e Architettura, Alma Mater Studiorum-Università di Bologna, 2015.

[BFB$^+$11] P. Banerjee, R. Friedrich, C. Bash, P. Goldsack, B. Huberman, J. Manley, C. Patel, P. Ranganathan, and A. Veitch. Everything as a service: Powering the new information economy. *Computer*, 44(3):36–43, March 2011.

[BFR99] Krysia Broda, Marcelo Finger, and Alessandra Russo. Labelled natural deduction for substructural logics. *Logic Journal of the IGPL*, 7(3):283–318, 1999.

[BG89] Antonio Brogi and Roberto Gorrieri. A distributed, net oriented semantics for Delta Prolog. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT '89: Proceedings of the International Joint Conference on Theory and Practice of Software Development Barcelona, Spain, March 13–17, 1989*, volume 351 of *Lecture Notes in Computer Science*, pages 162–177. Springer Berlin Heidelberg, Barcelona, Spain, 1989.

[BGLR02] Krysia Broda, Dov M. Gabbay, Luís C. Lamb, and Alessandra Russo. Labelled natural deduction for conditional logics of normality. *Logic Journal of the IGPL*, 10(2):123–163, 2002.

[Bha01] Ganesh D. Bhatt. Knowledge management in organizations: Examining the interaction between technologies, techniques, and people. *Journal of Knowledge Management*, 5(1):68–75, 2001.

[BHC98] Chumki Basu, Haym Hirsh, and William Cohen. Recommendation as classification: Using social and content-based information in recommendation. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 714–720, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[BJW00] Claudio Bettini, Sushil G. Jajodia, and Sean X. Wang. *Time Granularities in Databases, Data Mining and Temporal Reasoning.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 2000.

[Bla00] Patrick Blackburn. Internalizing labelled deduction. *Journal of Logic and Computation*, 10(1):137–168, 2000.

[BLM94] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *Journal of Logic Programming*, 19–20:443–502, 1994. Special Issue: Ten Years of Logic Programming.

[BP80] J. Barwise and J. Perry. *The Situation Underground.* Stanford University Press, 1980.

[BPRD08] Mike Botts, George Percivall, Carl Reed, and John Davidson. *OGC® Sensor Web Enablement: Overview and High Level Architecture*, pages 175–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[Bra13] Max Bramer. *Logic Programming with Prolog.* Springer, 2nd edition, 2013.

[Bro91] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1):139 – 159, 1991.

[Bro11] Jason Brownlee. *Clever algorithms: nature-inspired programming recipes.* Jason Brownlee, 2011.

[BRS07] Gerhard Brewka, Floris Roelofsen, and Luciano Serafini. Contextual default reasoning. In *IJCAI*, pages 268–273, 2007.

[Car15] Matteo Carano. Sperimentazione di tecnologie raspberry in contesti di home intelligence. Bachelor's Thesis – Scuola di Ingegneria e Architettura, Alma Mater Studiorum-Università di Bologna, 2015.

[Cas98] Cristiano Castelfranchi. Modelling social action for AI agents. *Artificial Intelligence*, 103(1-2):157–182, August 1998.

[Cas12] Cristiano Castelfranchi. Goals, the true center of cognition. In Fabio Paglieri, Luca Tummolini, Rino Falcone, and Maria Miceli, editors, *The Goals of Cognition. Essays in Honor of Cristiano Castelfranchi*, Tributes, chapter 41, pages 837–882. College Publications, December 2012.

[CCC$^+$04] V. Callaghan, G. Clarke, M. Colley, H. Hagras, J. S. Y. Chin, and F. Doctor. Inhabited intelligent environments. *BT Technology Journal*, 22(3):233–247, Jul 2004.

[CCMSP93] E. Ciapessoni, E. Corsetti, A. Montanari, and P. San Pietro. Embedding time granularity in a logical specification language for synchronous real-time systems. *Sci. Comput. Program.*, 20(1-2):141–171, April 1993.

[CD16a] Roberta Calegari and Enrico Denti. Building Smart Spaces on the Home Manager platform. *ALP Newsletter*, December 2016.

[CD16b] Roberta Calegari and Enrico Denti. The Butlers framework for socio-technical smart spaces. In Franco Bagnoli, Anna Satsiou, Ioannis Stavrakakis, Paolo Nesi, Giovanna Pacini, Yanina Welp, Thanassis Tiropanis, and Dominic DiFranzo, editors, *Internet Science. 3rd International Conference (INSCI 2016)*, volume 9934 of *LNCS*, pages 306–317. Springer, 2016.

[CD18] Roberta Calegari and Enrico Denti. *Context Reasoning and Prediction in Smart Environments: The Home Manager Case*, pages 451–460. Springer International Publishing, Cham, 2018. Proceedings of IIMSS 2017, Vilamoura, Portugal, 21-23 June 2017.

[CDDO16] Roberta Calegari, Enrico Denti, Agostino Dovier, and Andrea Omicini. Labelled variables in logic programming: Foundations. In Camillo Fiorentini and Alberto Momigliano, editors, *CILC 2016 – Italian Conference on Computational Logic*, volume 1645 of *CEUR Workshop Proceedings*, pages 5–20, Milano, Italy, 20-22 June 2016. CEUR-WS. Proceedings of the 31st Italian Conference on Computational Logic.

[CDDO17] Roberta Calegari, Enrico Denti, Agostino Dovier, and Andrea Omicini. Extending logic programming with labelled variables: Model and semantics. *Fundamenta Informaticae*, 2017. Special Issue CILC 2016.

[CDGA10] Yinong Chen, Zhihui Du, and Marcos García-Acosta. Robot as a service in cloud computing. In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pages 151–158. IEEE, 2010.

[CDMO16] Roberta Calegari, Enrico Denti, Stefano Mariani, and Andrea Omicini. Towards logic programming as a service: Experiments in tuProlog. In Corrado Santoro, Fabrizio Messina, and Massimiliano De Benedetti, editors, *WOA 2016 – 17th Workshop "From Objects to Agents"*, volume 1664 of *CEUR Workshop Proceedings*, pages 91–99. Sun SITE Central Europe, RWTH Aachen University, 29–30 July 2016. Proceedings of the 17th Workshop "From Objects to Agents" co-located with 18th European Agent Systems Summer School (EASSS 2016).

[CDMO17] Roberta Calegari, Enrico Denti, Stefano Mariani, and Andrea Omicini. Logic Programming as a Service (LPaaS): Intelligence for the IoT. In Giancarlo Fortino, MengChu Zhou, Zofia Lukszo, Athanasios V. Vasilakos, Francesco Basile, Carlos Palau, Antonio Liotta, Maria Pia Fanti, Antonio Guerrieri, and Andrea Vinci, editors, *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC 2017)*. IEEE, May 2017.

[CDO15] Roberta Calegari, Enrico Denti, and Andrea Omicini. Labelled variables in logic programming: A first prototype in tuProlog. In Elena Bellodi and Alessio Bonfietti, editors, *Proceedings of the Doctoral Consortium of the 14th Symposium of the Italian Association for Artificial Intelligence (AI*IA 2015 DC)*, volume 1485 of *CEUR Workshop Proceedings*, pages 25–30, Ferrara, Italy, 23–24 September 2015. AI*IA, CEUR-WS.

[CEF$^+$12] Marie Chan, Daniel EstèVe, Jean-Yves Fourniols, Christophe Escriba, and Eric Campo. Smart wearable systems: Current status and future challenges. *Artif. Intell. Med.*, 56(3):137–156, November 2012.

[Cel15] Alessandro Celi. Smart home: reasoning e proattività applicate ad un caso di studio. Bachelor's Thesis – Scuola di Ingegneria e Architettura, Alma Mater Studiorum-Università di Bologna, 2015.

[CF07] André C. M. Costa and Gonçalves P. Filho. Cores: Contextaware, ontology-based recommender system for service recommendation. In *Proc. CAiSE'07 Workshop on Ubiquitous Mobile Information and Collaboration Systems*, 2007.

[CF14] Mats Carlsson and Thom Fruehwirth. *Sicstus PROLOG User's Manual 4.3*. Books On Demand - Proquest, 2014.

[CFJ03] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, 18(3):197–207, September 2003.

[CFJ04a] H. Chen, T. Finin, and A. Joshi. Semantic web in the context broker architecture. In *Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. Proceedings of the*, pages 277–286, March 2004.

[CFJ$^+$04b] H. Chen, T. Finin, A. Joshi, L. Kagal, F. Perich, and D. Chakraborty. Intelligent agents meet the semantic web in smart spaces. *Internet Computing, IEEE*, 8(6):69–79, Nov 2004.

[CFP89] José C. Cunha, Maria C. Ferreira, and Luís Moniz Pereira. Programming in Delta Prolog. In Giorgio Levi and Maurizio Martelli, editors, *Logic Programming, Proceedings of the 6th International Conference (ICLP 1989), Lisbon, Portugal, 19-23 June 1989*, pages 487–504, Cambridge, MA, USA, 19–23 June 1989. MIT Press.

[CG81] Keith L. Clark and Steve Gregory. A relational language for parallel programming. In *1981 Conference on Functional Programming Languages and Computer Architecture (FPCA '81)*, pages 171–178, New York, NY, USA, 1981. ACM.

[Che05] Annie Chen. Context-aware collaborative filtering system: Predicting the user&#39;s preference in the ubiquitous computing environment. In *Proceedings of the First International Conference on Location- and Context-Awareness*, LoCA'05, pages 244–253, Berlin, Heidelberg, 2005. Springer-Verlag.

[CKLM91] S. E. Conry, K. Kuwabara, V. R. Lesser, and R. A. Meyer. Multistage negotiation for distributed constraint satisfaction. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1462–1477, Nov 1991.

[Cla87] Keith L. Clark. PARLOG: The language and its applications. In Jacobus W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe. Volume II: Parallel Languages. Eindhoven, The Netherlands, 15–19 June 1987. Proceedings*, volume 259 of *Lecture Notes in Computer Science*, pages 30–53. Springer, Berlin, Heidelberg, 1987.

[Cla97] William J. Clancey. *Situated Cognition: On Human Knowledge and Computer Representations*. Cambridge University Press, New York, NY, USA, 1997.

[Cle88] J. C. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, July 1988.

[CMS$^+$17] Davide Calvaresi, Mauro Marinoni, Arnon Sturm, Michael Schumacher, and Giorgio Buttazzo. The challenge of real-time multi-agent systems for enabling iot and cps. In *Proceedings of the International Conference on Web Intelligence*, WI '17, pages 356–364, New York, NY, USA, 2017. ACM.

[CNW12] L. Chen, C. D. Nugent, and H. Wang. A knowledge-driven approach to activity recognition in smart homes. *IEEE Transactions on Knowledge and Data Engineering*, 24(6):961–974, June 2012.

[CO09] Matteo Casadei and Andrea Omicini. Situated tuple centres in ReSpecT. In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, volume III, pages 1361–1368. ACM, 8–12 March 2009.

[CO10] Matteo Casadei and Andrea Omicini. Programming agent-environment interaction for mas situatedness in respect. *The Knowledge Engineering Review*, January 2010.

[Coh90] Jacques Cohen. Constraint Logic Programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[Col90] Alain Colmerauer. An introduction to prolog iii. In John W. Lloyd, editor, *Computational Logic. Symposium Proceedings, Brussels, November 13/14, 1990*, ESPRIT Basic Research Series, pages 37–79. Springer, 1990.

[Cos03] P Dockhorn Costa. Towards a services platform for context-aware applications. *Telem-atics, University of Twente, Enschede. Netherlands. August*, 2003.

[CR87] James Clifford and A. Rao. A simple, general structure for temporal domains. In *Temporal Aspects in Information Systems, Proceedings of the IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems, Sophia-Antipolis, France, 13-15 May, 1987*, pages 17–28, 1987.

[CR04] Carlo Combi and Rosalba Rossato. *Temporal Functional Dependencies with Multiple Granularities: A Logic Based Approach*, pages 864–873. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[Cre93] Daniel Crevier. *AI: The tumultuous history of the search for artificial intelligence.* Basic Books, 1993.

[Cus10] Michael Cusumano. Cloud computing and SaaS as new computing platforms. *Communications of the ACM*, 53(4):27–29, April 2010.

[CV08] Maurizio Cimadamore and Mirko Viroli. Integrating Java and Prolog through generic methods and type inference. In *Proc. ACM SAC 2008*, pages 198–205, 2008.

[Dav02] Andrew Davison. Logic programming languages for the internet. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, pages 66–104, London, UK, UK, 2002. Springer-Verlag.

[DC05] Sajal K. Das and Diane J. Cook. Designing smart environments: A paradigm based on learning and prediction. In *Proceedings of the First International Conference on Pattern Recognition and Machine Intelligence*, PReMI'05, pages 80–90, Berlin, Heidelberg, 2005. Springer-Verlag.

[DC15] Enrico Denti and Roberta Calegari. Butler-ising HomeManager: A pervasive multi-agent system for home intelligence. In Stephane Loiseau, Joaquim Filipe, Beatrice Duval, and Jaap Van Den Herik, editors, *7th Int. Conf. on Agents and Artificial Intelligence (ICAART 2015)*, pages 249–256, Lisbon, Portugal, 10–12 January 2015. SCITEPRESS.

[dCFLHRM10] Luis M. de Campos, Juan M. Fernández-Luna, Juan F. Huete, and Miguel A. Rueda-Morales. Combining content-based and collaborative recommendations: A hybrid approach based on bayesian networks. *Int. J. Approx. Reasoning*, 51(7):785–799, September 2010.

[DCP14] Enrico Denti, Roberta Calegari, and Marco Prandini. Extending a smart home multi-agent system with role-based access control. In *5th Int. Conf. on Internet Tech & Society*, pages 23–30, Taipei, Taiwan, 10–12 December 2014. IADIS Press. Best Paper Award.

[DCT07] Benjamin Devèze, Caroline Chopinaud, and Patrick Taillibert. *ALBA: A Generic Library for Programming Mobile Agents with Prolog*, pages 129–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[DDC96] Pierre Deransart, AbdelAli Ed Dbali, and Laurent Cervoni. *Prolog: The Standard. Reference Manual.* Springer, 1996.

[DDF$^+$06] Simon Dobson, Spyros Denazis, Antonio Fernandez, Dominique Gaiti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, 12 2006.

[Den14] Enrico Denti. Novel pervasive scenarios for home management: the Butlers architecture. *SpringerPlus*, 3(52):1–30, January 2014.

[Dey01] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, January 2001.

[DFP13] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Autonomous agents coordination: Action languages meet CLP(FD) and Linda. *Theory and Practice of Logic Programming*, 13(2):149–173, September 2013.

[DGB99] Marcello D'Agostino, Dov M. Gabbay, and Krysia Broda. Tableau methods for substructural logics. In Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*, pages 397–467. Springer Netherlands, Dordrecht, 1999.

[DHCL04] F. Doctor, H. Hagras, V. Callaghan, and A. Lopez. An adaptive fuzzy learning mechanism for intelligent agents in ubiquitous computing environments. In *Proceedings World Automation Congress, 2004.*, volume 16, pages 101–106, June 2004.

[DLC87] E. H. Durfee, V. R. Lesser, and D. D. Corkill. Coherent cooperation among communicating problem solvers. *IEEE Trans. Comput.*, 36(11):1275–1291, November 1987.

[DMP93] Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps. Design and implementation of the glue-nail database system. *SIGMOD Rec.*, 22(2):147–156, June 1993.

[DO01] Enrico Denti and Andrea Omicini. LuCe: A tuple-based coordination infrastructure for Prolog and Java agents. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):139–141, March-June 2001.

[DOC13] Enrico Denti, Andrea Omicini, and Roberta Calegari. tuProlog: Making Prolog ubiquitous. *ALP Newsletter*, October 2013.

[DOR01] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 184–198. Springer, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[DOR05] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Science of Computer Programming*, 57(2):217–250, August 2005.

[DP09] Agostino Dovier and Enrico Pontelli. Present and future challenges for ASP systems. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning. 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 622–624. Springer, 2009.

[DST98] Pierangelo Dell'Acqua, Fariba Sadri, and Francesca Toni. *Combining Introspection and Communication with Rationality and Reactivity in Agents*, pages 17–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[Dug12] Dominic Duggan. *Service-Oriented Architecture*, pages 207–358. John Wiley & Sons, Inc., 2012.

[DY06] Simon Dobson and Juan Ye. Using fibrations for situation identification. *Pervasive 2006 workshop proceedings*, 01 2006.

[DYZ09] Mieso K. Denko, Laurence Tianruo Yang, and Yan Zhang. *Autonomic Computing and Networking.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[ECB06] Richard Etter, Patricia Dockhorn Costa, and Tom Broens. A rule-based approach towards context-aware user notification services. In *2006 ACS/IEEE International Conference on Pervasive Services (ICPS 2006)*, pages 281–284, Lyon, France, 26–29 June 2006. IEEE.

[EJB17] EJB. Home Page. http://www.oracle.com/technetwork/java/javaee/ejb/, 2017.

[EM14] M. C. Eddin and Z. Mammeri. Non-functional properties aware configuration selection in component-based systems. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–7, June 2014.

[EMS+04] Ulle Endriss, Paolo Mancarella, Fariba Sadri, Giacomo Terreni, and Francesca Toni. Abductive logic programming with CIFF: System description. In Jóse Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 680–684. Springer, 2004.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall / Pearson Education International, Upper Saddle River, NJ, USA, 2005.

[Fam15] Bob Familiar. *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions.* Apress, Berkely, CA, USA, 1st edition, 2015.

[FB98] B Feijó and J Bento. A logic-based environment for reactive agents in intelligent cad systems. *Advances in Engineering Software*, 29(10):825–832, 1998.

[Fel06] Christiane Fellbaum. Wordnet(s). In Keith Brown, editor, *Encyclopedia of Language and Linguistics*, volume 13, pages 665–670. Elsevier, 2nd edition, 2006.

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

[FGRS14] Giancarlo Fortino, Antonio Guerrieri, Wilma Russo, and Claudio Savaglio. Integration of agent-based and Cloud Computing for the smart objects-oriented IoT. In *2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 493–498, May 2014.

[FIP02] Foundation for Intelligent Physical Agents (FIPA). *Agent Communication Language Specifications*, 2002.

[FJK$^+$01] Tim Finin, Anupam Joshi, Lalana Kagal, Olga Ratsimore, Vlad Korolev, and Harry Chen. Information agents for mobile and embedded devices. In Matthias Klusch and Franco Zambonelli, editors, *Cooperative Information Agents V. 5th InternationalWorkshop (CIA 2001), Modena, Italy, 6–8 May 2001. Proceedings*, pages 264–286. Springer, Berlin, Heidelberg, 2001.

[FK97] Tze Ho Fung and Robert Kowalski. The iff proof procedure for abductive logic programming. *The Journal of Logic Programming*, 33(2):151–165, 1997.

[FN99] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, September 1999.

[FN08] Kary Främling and Jan Nyman. Information architecture for intelligent products in the internet of things. *Beyond Business Logistics proceedings of NOFOMA*, pages 224–229, 2008.

[FS97] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, October 1997.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, May 2002.

[Gab96] Dov M. Gabbay. *Labelled Deductive Systems, Volume 1*, volume 33 of *Oxford Logic Guides*. Clarendon Press, September 1996.

[GBH$^+$05] M. Grossmann, M. Bauer, N. Honle, U. P. Kappeler, D. Nicklas, and T. Schwarz. Efficiently managing context information for large-scale scenarios. In *Third IEEE International Conference on Pervasive Computing and Communications*, pages 331–340, March 2005.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GG01] Chiara Ghidini and Fausto Giunchiglia. Local models semantics, or contextual reasoning=locality+compatibilitythis paper is a substantially revised and extended version of a paper with the same title presented at the 1998 knowledge representation and reasoning conference (kr'98). the order of the names is alphabetical. *Artificial Intelligence*, 127(2):221 – 259, 2001.

[GH90] Les Gasser and Michael N. Huhns, editors. *Distributed Artificial Intelligence: Vol. 2.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[GHM$^+$08] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. Owl 2: The next step for owl. *Web Semant.*, 6(4):309–322, November 2008.

[Giu92] Fausto Giunchiglia. Contextual reasoning. *EPISTEMOLOGIA, SPECIAL ISSUE ON I LINGUAGGI E LE MACCHINE*, 345:345–364, 1992.

[Goo14a] Google. Works with Nest. http://techcrunch.com/2014/06/23/google-makes-its-nest-at-the-center-of-the-smart-home/, 2014.

[Goo14b] Google Meteo. http://www.androidcentral.com/google-now, 2014.

[Got97] Linda S. Gottfredson. Why g matters: The complexity of everyday life. *Intelligence*, 24(1):79 – 132, 1997. Special Issue Intelligence and Social Policy.

[Gro] GrovePi Home. http://www.dexterindustries.com/grovepi/.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, June 1993.

[GS12] Gudni Gudnason and Raimar Scherer. *eWork and eBusiness in Architecture, Engineering and Construction: ECPPM 2012.* CRC Press, 2012.

[GSB02] Hans W. Gellersen, Albercht Schmidt, and Michael Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mob. Netw. Appl.*, 7(5):341–351, October 2002.

[GSK$^+$04] Mahmoud Ghorbel, Maria-Teresa Segarra, Jérome Kerdreux, Ronan Keryell, Andre Thepaut, and Mounir Mokhtari. *Networking and Communication in Smart Home for People with Disabilities*, pages 937–944. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[GSO] GSON. Home Page. 2017.

[GWPZ04] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. An ontology-based context model in intelligent environments. In *IN PROCEEDINGS OF COMMUNICATION NETWORKS AND DISTRIBUTED SYSTEMS MODELING AND SIMULATION CONFERENCE*, pages 270–275, 2004.

[GWT$^+$09] Tao Gu, Zhanqing Wu, Xianping Tao, Hung Keng Pung, and Jian Lu. epsicar: An emerging patterns based approach to sequential, interleaved and concurrent activity recognition. In *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*, PERCOM '09, pages 1–9, Washington, DC, USA, 2009. IEEE Computer Society.

[HB] Sinja Helfenstein and Edoardo Beutler. Context-awareness seminar context-aware computing. Seminar.

[HBKR10] Jomi F. Hübner, Olivier Boissier, Rosine Kitio, and Alessandro Ricci. Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous Agents and Multi-Agent Systems*, 20(3):369–400, May 2010.

[HCF03] Henry Hexmoor, Cristiano Castelfranchi, and Rino Falcone, editors. *Agent Autonomy*, volume 7 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer US, Boston, MA, USA, 2003.

[Hew90] Carl Hewitt. *The Foundation of Artificial Intelligence—a Sourcebook*, chapter The Challenge of Open Systems, pages 383–395. Cambridge University Press, New York, NY, USA, 1990.

[HHC93] Philip Husbands, Inman Harvey, and Dave Cliff. An evolutionary approach to situated ai. In *Proceedings of the 9th Bi-annual Conference of the Society for the Study of Artificial Intelligence and the Simulation of Behaviour (AISB 93)*, pages 61–70, 1993.

[HI04] K. Henricksen and J. Indulska. Modelling and using imperfect context information. In *IEEE Annual Conference on Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second*, pages 33–37, March 2004.

[HI06] Karen Henricksen and Jadwiga Indulska. Developing context-aware pervasive computing applications: Models and approach. *Pervasive Mob. Comput.*, 2(1):37–64, February 2006.

[HIR02] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. *Modeling Context Information in Pervasive Computing Systems*, pages 167–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[HJ96] Horst Hendriks-Jansen. *Catching Ourselves in the Act: Situated Activity, Interactive Emergence, Evolution, and Human Thought.* MIT Press, Cambridge, MA, USA, 1st edition, 1996.

[HM10] Terry Halpin and Tony Morgan. *Information modeling and relational databases.* Morgan Kaufmann, 2010.

[HMB+94] J. C. T. Hallam, C. A. Malcolm, M. Brady, R. Hudson, and D. Partridge. Behaviour: Perception, action and intelligence - the view from situated robotics [and discussion]. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 349(1689):29–42, 1994.

[Hol92] Christian Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 260–268. Springer, 1992.

[Hom14] Home Manager. http://apice.unibo.it/xwiki/bin/view/Products/HomeManager, 2014.

[HPSvH03] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From shiq and rdf to owl: the making of a web ontology language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7 – 26, 2003.

[HPT97] Michael Hawley, R Dunbar Poor, and Manish Tuteja. Things that think. *Personal Technologies*, 1(1):13–20, 1997.

[HWD12] Bo Hu, Zhixue Wang, and Qingchao Dong. A modeling and reasoning approach using description logic for context-aware pervasive computing. In Jingsheng Lei, Fu Lee Wang, Hepu Deng, and Duoqian Miao, editors, *Emerging Research in Artificial Intelligence and Computational Intelligence: International Conference, AICI 2012, Chengdu, China, October 26-28, 2012. Proceedings*, volume 315 of *Communications in Computer and Information Science*, pages 155–165. Springer, Berlin, Heidelberg, 2012.

[Ibr16] Dogan Ibrahim. An overview of soft computing. *Procedia Computer Science*, 102(Supplement C):34 – 38, 2016. 12th International Conference on Application of Fuzzy Systems and Soft Computing, ICAFS 2016, 29-30 August 2016, Vienna, Austria.

[ICW93] Jean-Louis Imbert, Jacques Cohen, and Marie-Dominique Weeger. An algorithm for linear constraint solving: Its incorporation in a Prolog meta-interpreter for CLP. *The Journal of Logic Programming*, 16(3):235–253, 1993.

[J2E17] J2EE. Home Page, 2017.

[JAD] JADE API. Home Page.

[Jav17] Java Persistence API. Home Page. http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html, 2017.

[Jer17] Jersey. Home Page. http://jersey.java.net, 2017.

[JF88] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19–20, Supplement 1:503–581, May–July 1994. Special Issue: Ten Years of Logic Programming.

[jos17] jose.4.j. Home Page. http://bitbucket.org/b_c/jose4j/, 2017.

[JSO17] JSON. Home Page, 2017.

[JXJ$^+$15] A. Jian, G. Xiaolin, Y. Jianwei, S. Yu, and H. Xin. Mobile crowd sensing for internet of things: A credible crowdsourcing model in mobile-sense service. In *2015 IEEE International Conference on Multimedia Big Data*, pages 92–99, April 2015.

[Kar05] Kenneth Karta. An investigation on personalized collaborative filtering for web service selection. *Honours Programme thesis, University of Western Australia, Brisbane*, 2005.

[Ken97] Carolyn M Hall Rosalind Kent. *ENCYCLOPEDIA OF MICROCOMPUTERS*, 1997.

[Ken06] James Kennedy. *Swarm Intelligence*, pages 187–219. Springer US, Boston, MA, 2006.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, July 1995.

[KM88] Kenneth M. Kahn and Mark S. Miller. Language Design and Open Systems. In Bernardo A. Huberman, editor, *Ecology of Computation.* Elsevier Science Publishers, North-Holland, 1988.

[KM03] Eleftheria Katsiri and Alan Mycroft. Knowledge representation and scalable abstract reasoning for sentient computing using first-order logic. In *Proceedings of Challenges and Novel Applications for Automatic Reasoning (CADE-19)*, pages 73–87, 2003.

[Kow83] Robert Kowalski. Logic programming. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23*, pages 133–145. North-Holland/IFIP, 1983.

[Kow85] R. Kowalski. *Logic-based Open Systems.* Department of Computing, Imperial College of Science and Technology, 1985.

[KRW$^+$04] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M.H. Butler, and L. Tran. Composite capability/preference profiles (cc/pp): Structure and vocabularies 1.0, w3c recommendation. Technical report, W3C, Jan 2004.

[KS96] Robert Kowalski and Fariba Sadri. *Towards a unified agent architecture that combines rationality with reactivity*, pages 135–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[KS99] Robert Kowalski and Fariba Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3):391–419, Nov 1999.

[KSL12] Ohbyung Kwon, Jae Moon Shim, and Geunchan Lim. Single activity sensor-based ensemble analysis for health monitoring of solitary elderly people. *Expert Syst. Appl.*, 39(5):5774–5783, April 2012.

[Li01] Xining Li. *IMAGO: A Prolog-based System for Intelligent Mobile Agents*, pages 21–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[LKRF99] Alexander Leonhardi, Uwe Kubach, Kurt Rothermel, and Andreas Fritz. Virtual information towers-a metaphor for intuitive, location-aware information access in a mobile environment. In *Proceedings of the 3rd IEEE International Symposium on Wearable Computers*, ISWC '99, pages 15–, Washington, DC, USA, 1999. IEEE Computer Society.

[Llo12] John W Lloyd. *Foundations of logic programming.* Springer Science & Business Media, 2012.

[LMMZ17] Marco Lippi, Marco Mamei, Stefano Mariani, and Franco Zambonelli. Coordinating distributed speaking objects. In *37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017).* IEEE Computer Society, 2017.

[Lok04] Seng W. Loke. Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. *The Knowledge Engineering Review*, 19(3):213–233, September 2004.

[Mar16a] Stefano Mariani. *Coordination of Complex Sociotechnical Systems: Self-organisation of Knowledge in* MoK. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer International Publishing, 1st edition, December 2016.

[Mar16b] Luca Marzaduri. Windows 10 IoT su Raspberry Pi 2: multi-paradigm programming tra Java e C#, 2016. Bachelor's Thesis – Scuola di Ingegneria e Architettura, Alma Mater Studiorum-Università di Bologna. `http://amslaurea.unibo.it/10299/`.

[MAT13] A. M. Mzahm, M. S. Ahmad, and A. Y. C. Tang. Agents of things (aot): An intelligent operational concept of the internet of things (iot). In *2013 13th International Conference on Intellient Systems Design and Applications*, pages 159–164, Dec 2013.

[MH06] P. Moore and B. Hu. A context framework for entity identification for the personalisation of learning in pedagogic systems. In *2006 10th International Conference on Computer Supported Cooperative Work in Design*, pages 1–6, May 2006.

[MH07] Philip Moore and Bin Hu. A context framework with ontology for personalised and cooperative mobile learning. In *Proceedings of the 10th International Conference on Computer Supported Cooperative Work in Design III*, CSCWD'06, pages 727–738, Berlin, Heidelberg, 2007. Springer-Verlag.

[MHH98] Chris Melhuish, Owen Holland, and Steve Hoddell. Collective sorting and segregation in robots with minimal sensing. In *Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior on From Animals to Animats 5*, pages 465–470, Cambridge, MA, USA, 1998. MIT Press.

[Mic15a] Microsoft. The internet of your things. https://dev.windows.com/en-us/iot/, 2015.

[Mic15b] Microsoft Projects. https://microsoft.hackster.io/en-US, 2015.

[MLdlR03] Miquel Montaner, Beatriz López, and Josep Lluís de la Rosa. A taxonomy of recommender agents on the internet. *Artificial Intelligence Review*, 19(4):285–330, Jun 2003.

[MMS$^+$12] Karandeep Malhi, Subhas Chandra Mukhopadhyay, Julia Schnepper, Mathias Haefke, and Hartmut Ewald. A zigbee-based wearable physiological parameters monitoring system. *IEEE sensors journal*, 12(3):423–430, 2012.

[MO15] Stefano Mariani and Andrea Omicini. Coordinating activities and change: an event-driven architecture for situated MAS. *Engineering Applications of Artificial Intelligence*, 41:298–309, May 2015.

[Mon84] Luis Monteiro. A proposal for distributed programming in logic. In J. A. Campbell, editor, *Implementations of Prolog*, Artificial Intelligence, pages 329–340. Ellis Horwood Limited, Chicester, UK, 1984.

[MPS$^+$16] Fabrizio Messina, Giuseppe Pappalardo, Corrado Santoro, Domenico Rosaci, and Giuseppe M. L. Sarné. A multi-agent protocol for service level agreement negotiation in cloud federations. *International Journal of Grid and Utility Computing*, 7(2):101–112, 2016.

[MQT17] MQTT. Home Page. http://mqtt.org, 2017.

[MRS04] Ian Millard, David De Roure, and Nigel Shadbolt. The use of ontologies in contextually aware environments. This workshop was held in conjunction with UbiComp 2004 Event Dates: September 2004, 2004.

[Nai88] Lee Naish. Parallelizing NU-Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the 5th International Conference and Symposium, Seattle, Washington, 15-19 August 1988*, pages 1546–1564, Cambridge, MA, USA, 1988. MIT Press.

[NB14] Grzegorz J. Nalepa and Szymon Bobek. Rule-based solution for context-aware reasoning on mobile devices. *Comput. Sci. Inf. Syst.*, 11(1):171–193, 2014.

[NH12] N. Noury and T. Hadidi. Computer simulation of the activity of the elderly person living independently in a health smart home. *Comput. Methods Prog. Biomed.*, 108(3):1216–1228, December 2012.

[Nie13] Gerrit Niezen. Ontologies for interaction: Enabling serendipitous interoperability in smart environments. *Journal of Ambient Intelligence and Smart Environments*, 5(1):135–137, January 2013.

[Nii86] H. Penny Nii. The blackboard model of problem solving and the evolution of blackboard architectures. *The AI Magazine*, 7(2):38–106, Summer 1986.

[NM01] Daniela Nicklas and Bernhard Mitschang. The nexus augmented world model: An extensible approach for mobile, spatially-aware applications. In *In: Wang, Yingxu (ed.); Patel, Shushma (ed.); Johnston, Ronald (ed.): Proceedings of the 7th International Conference on Object-Oriented Information Systems : OOIS '01 ; Calgary, Canada, August 27-29, 2001, pp. 392-401*, 01 2001.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, Inc., New York, NY, USA, 1989.

[OAu] OAuth. Home page. http://oauth.net.

[OD01a] Andrea Omicini and Enrico Denti. Formal ReSpecT. *Electronic Notes in Theoretical Computer Science*, 48:179–196, June 2001.

[OD01b] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[Omi02] Andrea Omicini. Towards a notion of agent coordination context. In Dan C. Marinescu and Craig Lee, editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Boca Raton, FL, USA, October 2002.

[Omi07] Andrea Omicini. Formal ReSpecT in the A&A perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97–117, June 2007.

[ON98] P. D. O'Brien and R. C. Nicol. Fipa — towards a standard for software agents. *BT Technology Journal*, 16(3):51–59, Jul 1998.

[OOR04] Andrea Omicini, Sascha Ossowski, and Alessandro Ricci. Coordination infrastructures in the engineering of multiagent systems. In Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter 14, pages 273–296. Kluwer Academic Publishers, June 2004.

[OP11] Mohammad Oliya and Hung Keng Pung. Towards incremental reasoning for context aware systems. In Ajith Abraham, Jaime Lloret Mauri, John F. Buford, Junichi Suzuki, and Sabu M. Thampi, editors, *Advances in Computing and Communications: First International Conference, ACC 2011, Kochi, India, July 22-24, 2011. Proceedings, Part I*, volume 190 of *Communications in Computer and Information Science*, pages 232–241. Springer, Berlin, Heidelberg, 2011.

[Ope14] OpenWeatherMap. https://openweathermap.org, 2014.

[ORV06] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. *Agens Faber*: Toward a theory of artefacts for MAS. *Electronic Notes in Theoretical Computer Science*, 150(3):21–36, 29 May 2006.

[ORV08] Andrea Omicini, Alessandro Ricci, and Giuseppe Vizzari. Smart environments as agent workspaces. *Ubiquitous Computing and Communication Journal*, CPE - Special Issue:95–104, June 2008. Special Issue on Coordination in Pervasive Environments.

[ORZ06] Andrea Omicini, Alessandro Ricci, and Nicola Zaghini. Distributed workflow upon linkable coordination artifacts. In Paolo Ciancarini and Herbert Wiklicky, editors, *Coordination Models and Languages*, volume 4038 of *LNCS*, pages 228–246. Springer, June 2006.

[OZ99a] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999. Special Issue: Coordination Mechanisms for Web Agents.

[OZ99b] Andrea Omicini and Franco Zambonelli. Tuple centres for the coordination of Internet agents. In *1999 ACM Symposium on Applied Computing (SAC'99)*, pages 183–190. ACM, 1999.

[Pao15] Daiana Paolini. Geolocalizzazione di servizi in un sistema di home intelligence. Bachelor's Thesis – Scuola di Ingegneria e Architettura, Alma Mater Studiorum-Università di Bologna, 2015.

[Par08] Lynne E. Parker. Distributed intelligence: Overview of the field and its application in multi-robot systems. *Journal of Physical Agents*, 2(1):5–14, 2008.

[Pas97] Jason Pascoe. The stick-e note architecture: Extending the interface beyond the user. In *Proceedings of the 2Nd International Conference on Intelligent User Interfaces*, IUI '97, pages 261–264, New York, NY, USA, 1997. ACM.

[Pay17] Payara. Home Page, 2017.

[POJ17] POJO. Home Page, 2017. https://www.martinfowler.com/bliki/POJO.html.

[PRJ12] S. Patidar, D. Rane, and P. Jain. A survey paper on cloud computing. In *2012 Second International Conference on Advanced Computing Communication Technologies*, pages 394–398, Jan 2012.

[PRT01] Agostino Poggi, Giovanni Rimassa, and Michele Tomaiuolo. Multi-user and security support for multi-agent systems. In *In: Proceedings of WOA 2001 (Dagli oggetti agli*, pages 13–18, 2001.

[PZCG14] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Sensing as a service model for smart cities supported by internet of things. *Transactions on Emerging Telecommunications Technologies*, 25(1):81–93, 2014.

[QY07] Hao Qin and Simon X. Yang. Adaptive neuro-fuzzy inference systems based approach to nonlinear noise cancellation for images. *Fuzzy Sets Syst.*, 158(10):1036–1063, May 2007.

[RAMC04] A. Ranganathan, J. Al-Muhtadi, and R. H. Campbell. Reasoning about uncertain contexts in pervasive computing environments. *IEEE Pervasive Computing*, 3(2):62–70, April 2004.

[Rao96] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John W. Perram, editors, *Agents Breaking Away*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, 22-25 January 1996, Proceedings.

[RC03] Anand Ranganathan and Roy H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, 7(6):353–364, December 2003.

[Rei78] Raymond Reiter. *On Closed World Data Bases*, pages 55–76. Springer US, Boston, MA, 1978.

[Rei84] Raymond Reiter. *Towards a Logical Reconstruction of Relational Database Theory*, pages 191–238. Springer New York, New York, NY, 1984.

[Ric16] Mark Richards. *Microservices AntiPatterns and Pitfalls.* O'Reilly, Sebastopol, CA, USA, 2016.

[Ris02] Hanna Risku. Situatedness in translation studies. *Cognitive systems research*, 3(3):523–533, 2002.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[Rob65] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Ros94] Michael A. Rosenman. Distributed artificial intelligence: Theory and praxis : N m avouris and l gasser (eds.). *Knowl.-Based Syst.*, 7:147–148, 1994.

[RPTC15] A. Ricci, M. Piunti, L. Tummolini, and C. Castelfranchi. The mirror world: Preparing for mixed-reality living. *Pervasive Computing, IEEE*, 14(2):60–63, Apr 2015.

[RRS11] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Introduction to Recommender Systems Handbook*, pages 1–35. Springer US, Boston, MA, 2011.

[RSS92] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Coral - control, relations and logic. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 238–250, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[RTA14] Carlo Ratti, Yaniv J. Turgeman, and Eric Alm. Smart toilets and sewer sensors are coming. *Wired*, March 2014.

[Rus96] Alessandra Maria Russo. *Modal Labelled Deductive Systems*. PhD thesis, Department of Computing, Imperial College London, UK, June 1996.

[RVO$^+$17] Alessandro Ricci, Mirko Viroli, Andrea Omicini, Stefano Mariani, Angelo Croatti, and Danilo Pianini. Spatial Tuples: Augmenting physical reality with tuple spaces. In Costin Badica, Amal El Fallah Seghrouchni, Aurélie Beynier, David Camacho, Cédric Herpson, Koen Hindriks, and Paulo Novais, editors, *Intelligent Distributed Computing X. Proceedings of the 10th International Symposium on Intelligent Distributed Computing – IDC 2016, Paris, France, October 10-12 2016*, volume 678 of *Studies in Computational Intelligence*, pages 121–130, Berlin, Heidelberg, 2017. Springer.

[SAFP15] Anastasios Skarlatidis, Alexander Artikis, Jason Filippou, and Georgios Paliouras. A probabilistic logic programming event calculus. *Theory and Practice of Logic Programming*, 15(2):213–245, March 2015. Special Issue on Probability, Logic and Learning.

[Sam15] Samsung Smart Things. https://www.smartthings.com, 2015.

[SAW94] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, WMCSA '94, pages 85–90, Washington, DC, USA, 1994. IEEE Computer Society.

[SC17] M. P. Singh and A. K. Chopra. The internet of things and multiagent systems: Decentralized intelligence in distributed computing. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1738–1747, June 2017.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, pages 434–441, New York, NY, USA, 1999. ACM.

[SGFW10] Harald Sundmaeker, Patrick Guillemin, Peter Friess, and Sylvie Woelfflé. Vision and challenges for realising the internet of things. *Cluster of European Research Projects on the Internet of Things, European Commision*, 3(3):34–36, 2010.

[SGRM12] NK Suryadevara, A Gaddam, RK Rayudu, and SC Mukhopadhyay. Wireless sensors network based safe home to care elderly people: Behaviour detection. *Sensors and Actuators A: Physical*, 186:277–283, 2012.

[SH12] Luciano Serafini and Martin Homola. Contextualized knowledge repositories for the semantic web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 12-13:64 – 87, 2012. Reasoning with context in the Semantic Web.

[Sha87] Ehud Y. Shapiro. *Concurrent Prolog – Vol. 1: Collected Papers.* Logic Programming. The MIT Press, Cambridge, MA, USA, 1987.

[SKP$^+$11] Hans Schaffers, Nicos Komninos, Marc Pallot, Brigitte Trousse, Michael Nilsson, and Alvaro Oliveira. Smart cities and the future internet: Towards cooperation frameworks for open innovation. In *The Future Internet*, pages 431–446. Springer-Verlag, 2011.

[SLP04] Thomas Strang and Claudia Linnhoff-Popien. A context modeling survey. In *In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, 2004.

[SM03] D. Saha and A. Mukherjee. Pervasive computing: a paradigm for the 21st century. *Computer*, 36(3):25–31, Mar 2003.

[SM15] Nagender Kumar Suryadevara and Subhas Chandra Mukhopadhyay. *Smart Homes: Design, Implementation and Issues.* Springer Publishing Company, Incorporated, 2015.

[Sma17] Paul Smart. Situating machine intelligence within the cognitive ecology of the internet. *Minds and Machines*, Online First:1–24, 2017.

[SP99] Perdita Stevens and Rob Pooley. *Using Uml: Software Engineering with Objects and Components*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[SSW86] Leon Sterling, Ehud Y. Shapiro, and David H.D. Warren. *The Art of Prolog. Advanced Programming Techniques*, volume 1994. MIT Press, 1986.

[STF08] Dairazalia Sánchez, Monica Tentori, and Jesús Favela. Activity recognition for the smart hospital. *IEEE Intelligent Systems*, 23(2):50–57, March 2008.

[Suc87] Lucy A. Suchman. *Plans and Situated Actions: The Problem of Human-machine Communication*. Cambridge University Press, New York, NY, USA, 1987.

[Sym94] Alex K. Sympson. *The Proof Theory and Semantics of Intuitionistic Modal Logics*. PhD thesis, University of Edinburgh, UK, 1994.

[Tar99] Paul Tarau. Jinni: Intelligent mobile agent programming at the intersection of java and prolog. In *In Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, 1999.

[Tre00] Shari Trewin. Knowledge-based recommender systems. *Encyclopedia of library and information science*, 69(Supplement 32):180, 2000.

[tuc08] tucson. Home page. http://tucson.apice.unibo.it/, 2008.

[tup01] tuprolog. Home page. tuprolog.apice.unibo.it, 2001.

[Twi] Twitter4J. Home page. http://twitter4j.org/en/index.html.

[Ued86] Kazunori Ueda. Guarded Horn clauses. In Eiiti Wada, editor, *Logic Programming '85. Proceedings of the 4th Conference Tokyo, Japan, 1–3 July 1985*, volume 221 of *Lecture Notes in Computer Science*, pages 168–179. Springer, Berlin, Heidelberg, 1986.

[UG96] Mike Uschold and Michael Gruninger. Ontologies: principles, methods and applications. *The Knowledge Engineering Review*, 11(2):93–136, 1996.

[van05] Mark van Setten. *Supporting People in Finding Information: Hybrid Recommender Systems and Goal-Based Structuring*. PhD thesis, Telematica Instituut, 12 2005.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[vSPK04] Mark van Setten, Stanislav Pokraev, and Johan Koolwaaij. *Context-Aware Recommendations in the Mobile Tourist Application COMPASS*, pages 235–244. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[VZ17] E. Vargiu and F. Zambonelli. Agent abstractions for engineering iot systems: A case study in smart healthcare. In *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*, pages 667–672, May 2017.

[WAKS97] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The zephyr abstract syntax description language. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, DSL'97, pages 17–17, Berkeley, CA, USA, 1997. USENIX Association.

[WDC$^+$04] Xiaohang Wang, Jin Song Dong, ChungYau Chin, SankaRavipriya Hettiarachchi, and Daqing Zhang. Semantic space: An infrastructure for smart spaces. *IEEE Pervasive Computing*, 3(3):32–39, 2004.

[Whi06] Brian Whitworth. Socio-technical systems. *Encyclopedia of human computer interaction*, pages 533–541, 2006.

[WML84] David A. Wolfram, Michael J. Maher, and Jean-Louis Lassez. A unified treatment of resolution strategies for logic programs. In Sten-Åke Tärnlund, editor, *2nd International Conference on Logic Programming (ICLP 1984)*, pages 263–276, Uppsala, Sweden, 2–6 July 1984.

[WMSC11] Hongyuan Wang, Rutvij Mehta, Sam Supakkul, and Lawrence Chung. Rule-based context-aware adaptation using a goal-oriented ontology. In *2011 International Workshop on Situation Activity & Goal Awareness (SAGAware '11)*, pages 67–76, New York, NY, USA, 2011. ACM.

[WSA$^+$95] R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. An overview of the parctab ubiquitous computing experiment. *IEEE Personal Communications*, 2(6):28–43, Dec 1995.

[Yah14] Yahoo Meteo. https://www.yahoo.com/news/weather, 2014.

[YCD08] Wan-Shiou Yang, Hung-Chi Cheng, and Jia-Ben Dia. A location-aware recommender system for mobile shopping environments. *Expert Systems with Applications*, 34(1):437 – 445, 2008.

[YKB$^+$03] Quanhe Yang, Muin J Khoury, Lorenzo Botto, JM Friedman, and W Dana Flanders. Improving the prediction of complex diseases by testing for multiple disease-susceptibility genes. *The American Journal of Human Genetics*, 72(3):636–649, 2003.

[YT03] Soe-Tsyr Yuan and Y.W. Tsao. A recommendation mechanism for contextualized mobile advertising. *Expert Systems with Applications*, 24(4):399 – 414, 2003.

[ZCM03] Alejandro Zunino, Marcelo Campo, and Cristian Mateos. Movilog: A platform for prolog-based strong mobile agents on the www. *Inteligencia artificial: Revista Iberoamericana de Inteligencia Artificial, ISSN 1137-3601, N. 21, 2003, pags. 83-92*, 7, 02 2003.

[ZO04] Franco Zambonelli and Andrea Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, November 2004. Special Issue: Challenges for Agent-Based Computing.

[ZOA$^+$15] Franco Zambonelli, Andrea Omicini, Bernhard Anzengruber, Gabriella Castelli, Francesco L. DeAngelis, Giovanna Di Marzo Serugendo, Simon Dobson, Jose Luis Fernandez-Marquez, Alois Ferscha, Marco Mamei, Stefano Mariani, Ambra Molesini, Sara Montagna, Jussi Nieminen, Danilo Pianini, Matteo Risoldi, Alberto Rosi, Graeme Stevenson, Mirko Viroli, and Juan Ye. Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive and Mobile Computing*, 17:236–252, February 2015. Special Issue "10 years of Pervasive Computing" In Honor of Chatschik Bisdikian.

List of Figures

List of Tables