



Nathalia Moraes do Nascimento

**Self-Configurable IoT Embedded Agents controlled by
Neural Networks**

Tese de Doutorado

Thesis presented to the Programa de Pós-graduação em Informática of PUC–Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Orientador : Prof. Carlos José Pereira de Lucena
Co-Orientador: Prof. Paulo Sérgio Conceição de Alencar

Rio de Janeiro
September 2019

Nathalia Moraes do Nascimento

**Self-Configurable IoT Embedded Agents controlled by
Neural Networks**

Thesis presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática. Approved by the Examination Committee.

Prof. Carlos José Pereira de Lucena

Advisor

Departamento de Informática — PUC-Rio

Prof. Bruno Feijó

Departamento de Informática –PUC-Rio

Prof. Markus Endler

Departamento de Informática –PUC-Rio

Prof. Claudia Maria Lima Werner

COPPE –UFRJ

Prof. Elder José Reoli Cirilo

Departamento de Computação –UFSJ

Rio de Janeiro, September 18 th, 2019

All rights reserved.

Nathalia Moraes do Nascimento

Nathalia Nascimento holds a Master degree in Informatics with emphasis in Software Engineering and Artificial Intelligence from Pontifical Catholic University of Rio de Janeiro (PUC-Rio). She has a Bachelor degree in Computer Engineering from State University of Feira de Santana (UEFS). She received a CAPES Scholarship to do part of her PhD at the University of Waterloo, Canada. Her main research topics are related to the areas of Software Engineering, Artificial Intelligence and Internet of Things. She is a scientpreneur (researcher-entrepreneur), being one of the founders of orientaMED, a company that is based on sensors and Artificial Intelligence.

Bibliographic data

Nascimento, Nathalia Moraes do

Self-Configurable IoT Embedded Agents controlled by Neural Networks / Nathalia Moraes do Nascimento; orientador: Carlos José Pereira de Lucena; co-orientador: Paulo Sérgio Conceição de Alencar. — 2019.

116 f. : il. (color); 30 cm

1. Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2019.

Inclui bibliografia.

1. Informática – Teses. 2. Agente Embarcado. 3. Agente baseado em variabilidade. 4. Agente Configurável. 5. Rede Neural Artificial. 6. Internet das Coisas (IoT). 7. Agentes IoT. I. Lucena, Carlos José Pereira de. II. Alencar, Paulo Sérgio Conceição de. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

Aknowledgments

To God, for giving me the opportunity to made my dream come true. I am really proud to be a PhD. As this “travel” is usually long and difficult, I know that I am lucky to have the following people with me, who I would like to thank:

- My family - Nilza, Geraldo, Gabriela, and Jullia - for the advices and encouragements, and for always supporting me in my decisions;
- My advisor, Professor Lucena, for the opportunity of learning with his experience and knowledge. It was an honor;
- Professor Paulo Alencar and Professor Donald Cowan, for receiving me as a Visitor Researcher at the University of Waterloo, discussing new ideas, motivating me to write papers, and reviewing all my documents, including this thesis;
- Juliano, for giving me the incentive to be more studious and organized;
- Vera, for being so helpful and for receiving all the Professor Lucena’s students with a big smile. Thank you for making my days at PUC happier;
- The staff members of the Department of Informatics, specially Regina Zanon, Cosme Leal and dona Angela, who are so helpful and patient with the students;
- My friends and colleagues, for making this journey less difficult.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This work was also supported by the Laboratory of Software Engineering (LES) at PUC-Rio. Our thanks also to CNPq, FAPERJ and PUC-Rio for their financial support.

Abstract

Nascimento, Nathalia Moraes do; Lucena, Carlos José Pereira de ; Alencar, Paulo Sérgio Conceição de . **Self-Configurable IoT Embedded Agents controlled by Neural Networks**. Rio de Janeiro, 2019. 116p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Agent-based Internet of Things (IoT) applications have recently emerged as applications that can involve sensors, wireless devices, machines and software that can exchange data and be accessed remotely. Such applications have been proposed in several domains including health care, smart cities and agriculture. Embodied Agents is a term used to denote intelligent embedded agents, which we use to design agents to the IoT domain. Each agent is provided with a ‘body’ that has sensors to collect data from the environment and actuators to interact with the environment, and a ‘controller’ that is usually represented by an artificial neural network. Because reconfigurable behavior is key for autonomous embodied agents, there is a spectrum of approaches to support system reconfigurations. However, there is a need for approaches to handle agents and environment variability, and for a broad spectrum of procedures to investigate the relationship between the body and the controller of an embodied agent, as the interaction between the agent and the environment changes. In addition to the body and controller variability of these agents, such as those variations related to the number and types of sensors as well as the number of layers and types of activation function for the neural network, it is also necessary to deal with the variability of the environment in which these agents are situated. A discussion of the embodied agents should have some formal basis in order to clarify these concepts. Notwithstanding, this thesis presents a reference model for self-configurable IoT embodied agents. Based on this reference model, we have created three approaches to design and test self-configurable IoT embodied agents: i) a software framework for the development of embodied agents to the Internet of Things (IoT) applications; ii) an architecture to configure the body and controller of the agents based on environment variants; and iii) a tool for testing embodied agents. To evaluate these approaches, we have conducted different case studies and experiments in different application domains.

Keywords

Embodied Agent; Variability-Awareness; Configurable Agent; Artificial Neural Network; Internet of Things (IoT); IoT Agents.

Resumo

Nascimento, Nathalia Moraes do; Lucena, Carlos José Pereira de; Alencar, Paulo Sérgio Conceição de. **Agentes Embarcados de IoT Auto-configuráveis controlados por Redes Neurais**. Rio de Janeiro, 2019. 116p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Aplicações em Internet das Coisas (IoT) baseadas em agentes têm surgido como aplicações que podem envolver sensores, dispositivos sem fio, máquinas e softwares que podem compartilhar dados e que podem ser acessados remotamente. Essas aplicações vêm sendo propostas em vários domínios de aplicação, incluindo cuidados em saúde, cidades inteligentes e agricultura. Uma terminologia comumente utilizada para representar agentes embarcados inteligentes é “embodied agents”, a qual é proposta nesse trabalho para projetar agentes para o domínio de IoT. Embodied agents significa agentes que possuem ‘corpo’, o qual pode ser definido pelos tipos de sensores e atuadores, e ‘controlador’, normalmente representada por uma rede neural artificial. Apesar da capacidade de reconfiguração ser essencial para embodied agents inteligentes, existem poucas tecnologias para suportar sistemas reconfiguráveis. Além disso, é necessário novas abordagens para lidar com as variabilidades dos agentes e do ambiente, e novos procedimentos para investigar a relação entre o corpo e o controlador de um embodied agent, assim como as interações entre as mudanças do agente e do ambiente. Além da variabilidade do corpo e do controlador desses agentes, a exemplo do número e tipos de sensores, assim como o número de camadas e tipos de função de ativação para a rede neural, também é preciso lidar com a variabilidade do ambiente em que esses agentes estão situados. A fim de entender melhor e esclarecer os conceitos de embodied agents, este trabalho apresenta um modelo de referência para embodied agents autoconfiguráveis de IoT. A partir desse modelo de referência, três abordagens foram criadas para projetar e testar agentes embarcados reconfiguráveis: i) um software framework para o desenvolvimento de embodied agents no domínio de internet das coisas; ii) uma arquitetura para configurar o corpo e controlador dos agentes de acordo com as variantes do ambiente; e iii) uma ferramenta para testar embodied agents. As abordagens foram avaliadas através de estudos de caso e experimentos em diferentes domínios de aplicação.

Palavras-chave

Agente Embarcado; Agente baseado em variabilidade; Agente Configurável; Rede Neural Artificial; Internet das Coisas (IoT); Agentes IoT.

Contents

1	Introduction	12
1.1	Problem Statement	13
1.2	Objectives	14
1.3	Research Questions	14
1.4	Structure of the Thesis	15
1.5	Contributions	15
1.6	Limitations	18
2	Background	19
2.1	Multiagent System	19
2.2	Embodied Agents	21
2.3	Evolutionary Algorithms	23
2.4	Artificial Neural Network (ANN)	24
2.5	Formal Methods	29
3	Related Work	33
3.1	Reference Models for Agents	33
3.2	Reference Models for Reconfigurable Systems	36
3.3	Approaches and Applications for Embodied Agents	37
4	Fundamentals of Reconfigurable Embodied Agents	39
4.1	Preliminary embodied agent concepts	39
4.2	Preliminary reconfiguration concept	41
4.3	Statecharts	41
5	Approaches and their Applications	50
5.1	Framework for IoT Embodied Agents	50
5.2	An Architecture for Embodied Agents Reconfiguration	76
5.3	An Approach to Test Embodied Agents	89
6	Conclusion	104
7	Future Work and Open Challenges	106
7.1	Morphology-based agent design	106
7.2	Descriptive Evaluations	107
7.3	Embodied agent testing and verification	107
8	Bibliography	108

List of Figures

2.1	The diagram of a generic agent provided by authors in (Russell and Norvig, 1995) [p.32].	20
2.2	The challenge: how does an embodied agent make decisions based on collected data?	21
2.3	A general embodied agent model.	22
2.4	Block diagram representation of nervous system (Haykin, 1994). Pg.28.	25
2.5	The model of a neuron (Haykin, 1994). P. 33.	26
2.6	Evolving a neural network. Adapted from Floreano and Mattiussi (2008, P. 317).	30
2.7	Example of orthogonal components. Adapted from Harel (1987, p.14).	31
4.1	Control loop executed by embodied agents.	40
4.2	General statechart of embodied agents - putting the main components together.	42
4.3	Body and Controller Configuration components of the embodied agent statechart.	43
4.4	Behavior component of the embodied agent statechart.	45
4.5	Statechart of Environment and Task Evaluation components.	46
4.6	The designed autonomous street lights exploiting the proposed reference model for the Body and Controller Configuration components.	47
4.7	The designed autonomous street lights exploiting the proposed reference model for the Agent's Behavior component.	49
5.1	An example of an IoT embodied agent.	51
5.2	An agent-based model to generate IoT applications.	52
5.3	Class diagram of FloT - Agents.	54
5.4	Class diagram of FloT - Behaviors.	55
5.5	Class diagram of FloT - Controllers.	56
5.6	An instance of FloT to create "Quantified Fruit."	58
5.7	Examples of scenarios.	61
5.8	Validation curve.	63
5.9	Learning curve.	63
5.10	Simulated Neighborhood.	65
5.11	Variables collected and set by streetlights.	66
5.12	A streetlight uses a neural network to make decisions based on collected data.	67
5.13	The neural network controller for smart street lights: zeroed weights (FloT's Application View).	67
5.14	Configuration file to evolve the neural network via genetic algorithm using FloT.	68
5.15	Simulation results - Most-Fit from each generation.	69

5.16	The Evolved Neural Network to be used as a controller for real Street Lights (FloT's Application View).	70
5.17	Prototyping the smart street light.	71
5.18	Real Scenario where we tested a network of three smart street lights prototypes.	74
5.19	Feature model of embodied agents' body.	78
5.20	Feature model of a embodied agents' controller.	79
5.21	High-level model of the self-configurable agent approach to generate embodied agents.	80
5.22	Schematic illustration of our architecture to store and retrieve machine learning models based on context. Published in (Nascimento et al., 2018).	82
5.23	Modeling variability-aware embodied agents.	84
5.24	Class diagram of a multiagent system composed of IoT embodied agents.	85
5.25	Training step: varying the background light context in a street light scenario to generate generic and specialized ML models.	87
5.26	Deployment step: selecting the trained model to use according to the context.	88
5.27	A Publish-Subscribe-based architecture to test MASs.	91
5.28	Testable Agent class.	92
5.29	Making FloT's agents as Testable Agents.	94
5.30	Setting log values for each Testable FloT agent.	94
5.31	Overview of the general application architecture.	95
5.32	Activity diagram of the streetlights.	96
5.33	Activity diagram of the ObserverAgent.	96
5.34	Simplified state machine for verifying test cases generated for the function "evaluate selected solution".	98
5.35	Simplified state machine for verifying test cases generated for the functions "switch the light ON" and "switch the light OFF".	100
5.36	Executing the state machine to test the function "switch the light ON": failure generated between states "switchLightON" and "detectLight" - specific log was not consumed.	100
5.37	Executing the state machine to test the evaluation solution: failure generated between states "calculatePeople" and "calculateTripDuration" - because the machine did not receive the log that indicates that everyone finished their routes during the selected solution.	101
5.38	Executing the state machine to test the evaluation solution.	101
5.39	Subscribing to receive only logs related to the evaluation solution testing.	101
5.40	Subscribing to receive logs from all agents.	101

List of Tables

5.1	Case I: Flexible Points	59
5.2	Configuration of experiments at Figure 5.7.	60
5.3	Subset of the training set.	62
5.4	Subset of the test set.	62
5.5	FloT's Flexible Points	64
5.6	Case I: Main statechart components.	76
5.7	Summary of embodied agents variability.	81
5.8	Street lights' performance results obtained while executing different models with different contexts.	88
5.9	Case II: Main statechart components.	89
5.10	Functional tests at different perspectives (Simplified Table)	99
5.11	Case III: Main statechart components.	102

...have the courage to follow your heart and intuition. They somehow already know what you truly want to become.

Steve Jobs

1

Introduction

Based on the Google Trends tool (Google, 2018), the Internet of Things (IoT) (Atzori et al., 2012) is emerging as a topic that is highly related to robotics and machine learning. In fact, the use of learning agents has been proposed as an appropriate approach to modeling IoT applications (Nascimento and Lucena, 2017). These types of applications address the problems of distributed control of devices that must work together to accomplish tasks (Atzori et al., 2012). This has caused agent-based IoT applications to be considered for several domains, such as health care, smart cities, and agriculture. For example, in a smart city, software agents can autonomously operate traffic lights (Nascimento and Lucena, 2017), driverless vehicles (Herrero-Perez and Martinez-Barbera, 2008) and street lights (Nascimento and Lucena, 2017).

Agents that can interact with other agents or the environment in which the applications are embedded are called *embodied agents* (Brooks, 1995; Marocco and Nolfi, 2007; Nolfi et al., 2016; Nascimento and Lucena, 2017). Examples of such agents can be found in areas such as autonomous robots and cyber-physical systems. The first step in creating an embodied agent is to design the agent's body, which determines the agent interaction with an application's sensors and actuators involving the signals that the agent will send and receive (Nolfi et al., 2016). As a second step, to sense the environment and behave accordingly, this agent is provided with a controller, that is usually represented by an artificial neural network (ANN).

However, deploying these applications in specific scenarios has been challenging because of the complex static and dynamic variability of the physical devices (e.g. sensors and actuators), the software scenario behavior and the environment. First, the physical devices may vary in terms of variables such as number and types of sensors, energy consumption, battery life and number and types of actuators (e.g. alarms). Second, the software scenario application behavior must vary in accordance with the variations in the physical devices and the environment. For example, this behavior may involve different types of communication and distinct forms of notification (e.g. alerts). Third, the deployment problem becomes even more complex as the environment brings

about variations in both the physical devices and the application behavior.

Reconfigurable behavior is fundamental to agents that deal with changing environments (Luckcuck et al., 2018), allowing the set of agents to be reconfigured to provide specialized and better solutions to specific environmental or soft-goal priorities changes (Lapouchnian et al., 2006). A reconfigurable system must have alternative ways to meet particular objectives, such as efficiency and precision, to be able to change configurations at runtime according to this specific criteria (MacDonald et al., 2004). There are some approaches that support the development of reconfigurable physical systems, but most of them are not based on a reference model that describes what the agents do, thus posing challenges for the development of robust and safe systems (Ingrand, 2019). A discussion of self-configurable embodied agents should have more clarifications, making possible the understanding of the relationship between the body and the controller of an embodied agent because the interactions between the agent and the environment change in complex ways.

This thesis presents a reference model for self-configurable IoT embodied agents based on statecharts (Harel, 1987) and, based on this model, provides three novel approaches to design and test self-configurable embodied agents: i) a software framework for the development of embodied agents to the Internet of Things (IoT) applications; ii) an architecture to configure the body and controller of the agents based on environment variants; and iii) a tool for testing embodied agents. To evaluate these approaches, we have conducted case studies and experiments in different application domains.

1.1

Problem Statement

Because reconfigurable behavior is key for autonomous embodied agents, there is a spectrum of approaches to support system reconfigurations (Yim et al., 2007; Luckcuck et al., 2018). We realized that there is a need for approaches to handle embodied agent and environment variability, and to investigate the relationship between the body and the controller of an embodied agent in terms of the complex and dynamic interactions between the agent and the environment. In addition to the body and controller variability of these agents, which involves, for example, the number and types of sensors as well as the number of layers and types of activation function for the neural network, it is also necessary to deal with the variability of the environment in which these agents are situated (Beer, 2008; Jelisavcic et al., 2018; Bredeche et al., 2018; Banarse et al., 2019).

1.2 Objectives

Our main objective in this thesis is to deliver feasible approaches to assist with the development of self-configurable IoT embodied agents. To this end, the thesis aims at:

- Identifying the main variation points of embodied agents that can be configured according to the environment changes;
- Providing a reference model for self-configurable IoT embodied agents;
- Developing an approach for designing embodied agents based on the proposed model;
- Developing an approach for testing embodied agents based on the proposed model;
- Developing application scenarios to validate the proposed software approaches.

1.3 Research Questions

To reach these objectives, we seek to answer the following four research questions:

- RQ1. How can embodied agents and their interactions with the environment be specified?
- RQ2. How to design and implement a software framework to support the development of embodied agents?
- RQ3. How to define an architecture to configure the body and controller of the agents based on the environment variability?
- RQ3.1. What is the variability related to embodied agents and how can this variability be represented?
- RQ3.2. How can we represent the effect of the relevant environment changes on the reconfiguration of the embodied agents?
- RQ4. How to design and implement an approach to test embodied agents and their variability?

The questions are in the order in which the answers will be provided in the text. For example, in chapter 4, we answer RQ1, and in Chapter 5, we answer RQ2, and so on.

1.4

Structure of the Thesis

In this section, we provide an overview of the thesis structure and summaries for each chapter.

Chapter 2 Background This chapter provides some necessary background information to allow the understanding of the thesis. In particular, we provide an overview about Multiagent Systems (MAS), Modeling Formalisms, and Artificial Neural Networks.

Chapter 3 Related Work We present the state of the art and the differences between this thesis and the works related to the formalizing and designing of reconfigurable embodied agents, including approaches and applications for embodied agents.

Chapter 4 Fundamentals of Reconfigurable Embodied Agents In this chapter, we cover the question **RQ1. How can embodied agents and their interactions with the environment be specified?**. First, we survey the main concepts of embodied agents requirements taking previously published work and personal experiences into account. Then, we use statecharts to provide a reference model for embodied agents and their interactions with the environment.

Chapter 5 Approaches and Applications This chapter describes the three novel approaches that we created to design and test self-configurable embodied agents based on our reference model. We present the specific problems that must be solved through these approaches and describe each of them in detail.

Chapters 6 and 7 Conclusions and Future Work We conclude by summarizing the thesis, highlighting the contributions and limitations, and proposing future work.

1.5

Contributions

We have investigated embodied agents for six years. As a result, our contributions are multi-fold:

- Fundamentals:
 1. A reference model for self-configurable embodied agents, representing the interactions between the agent and the environment changes.
- Approaches:

1. The Framework for the Internet of Things (FIoT): A novel software framework to instantiate different applications based on IoT embodied agents (Nascimento and Lucena, 2017). We introduced the concept of embodied agents into the Internet of Things domain;
 2. A context-aware approach to reconfigure embodied agents (Nascimento et al., 2018; Nascimento, 2018);
 3. A testing approach to evaluate embodied agents-based applications (Nascimento et al., 2017; Nascimento et al., 2019).
- Applications:
1. The design and implementation of different applications for each one of the approaches (Nascimento et al., 2015; Nascimento Marx Leles Viana, 2016; Nascimento and Lucena, 2017; Nascimento and Lucena, 2017).

In addition, we have produced 11 scientific articles and 4 technical reports that contributed to the development of this thesis.

1.5.1 Scientific Articles

2019

1. (submitted) Nascimento, N., Lucena, C., Alencar, P., and Viana, C. J. (2019). A Metadata-Driven Approach for Testing Self-Organizing Multiagent Systems. Submitted to ACM Transactions.

2018

1. Nascimento, N., Alencar, P., Lucena, C., and Cowan, D. (2018, December). An IoT Analytics Embodied Agent Model based on Context-Aware Machine Learning. In 2018 IEEE International Conference on Big Data (Big Data) (pp. 5170-5175). IEEE.
2. Nascimento, N., Alencar, P., Lucena, C., and Cowan, D. (2018, December). Toward Human-in-the-Loop Collaboration Between Software Engineers and Machine Learning Algorithms. In 2018 IEEE International Conference on Big Data (Big Data) (pp. 3534-3540). IEEE.
3. (Qualis B1) Nascimento, N, Paulo Alencar, Carlos Lucena, and Donald Cowan. **A context-aware machine learning-based approach**. In:

ACM. Computer Science and Software Engineering (CASCON), 28th Annual International Conference on. [S.l.], 2018.

4. (short paper - Doctoral Consortium) Nascimento, Nathalia. **A Self-Configurable IoT Agent System based on Environmental Variability**. Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). 2018.

2017

1. (Qualis A1) do Nascimento, Nathalia Moraes, and Carlos José Pereira de Lucena. **Fiot: An agent-based framework for self-adaptive and self-organizing applications based on the internet of things**. Information Sciences 378 (2017): 161-176.
2. (Qualis B1) do Nascimento, Nathalia Moraes, and Carlos José Pereira de Lucena. **Engineering cooperative smart things based on embodied cognition**. Adaptive Hardware and Systems (AHS), 2017 NASA/ESA Conference on. IEEE, 2017.
3. (Qualis B1) do Nascimento, N. M., Viana, C. J. M., von Staa, A., and Lucena, C. (2017). **A Publish-Subscribe based Architecture for Testing Multiagent Systems**. 29th International Conference on Software Engineering and Knowledge Engineering (SEKE'2017) (pp. 521-526).

2016

1. (short paper) do Nascimento, Nathalia Moraes, Marx Leles Viana, and Carlos Lucena. **An IoT-based Tool for Human Gas Monitoring**. IXV Congresso Brasileiro de Informatica em Saude (CBIS). Vol. 1. 2016.
2. (Qualis B1) Briot, Jean-Pierre, Nathalia Moraes de Nascimento, and Carlos Lucena. **A multi-agent architecture for quantified fruits: Design and experience**. 28th International Conference on Software Engineering and Knowledge Engineering (SEKE'2016). 2016.

2015

1. (Qualis B1) do Nascimento, Nathalia Moraes, Carlos Lucena, and Hugo Fuks. **Modeling quantified things using a multi-agent system**. 2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT). 2015.

1.5.2

Technical Reports

1. Nascimento, N., Lucena, C., Alencar, P., and Viana, C. J. (2019). Testing Self-Organizing Multiagent Systems. arXiv preprint arXiv:1904.01736.
2. Nascimento, N., Alencar, P., Lucena, C., and Cowan, D. (2018). **Machine Learning-based Variability Handling in IoT Agents**. arXiv preprint arXiv:1802.03858.
3. Nascimento, N., Lucena, C., Alencar, P., and Cowan, D. (2018). **Software Engineers vs. Machine Learning Algorithms: An Empirical Study Assessing Performance and Reuse Tasks**. arXiv preprint arXiv:1802.01096.
4. Nascimento, N., Lucena, C., Fuks, H. (2015). **Internet das Coisas para Conservação de Frutas: O Caso da Banana**. Serie Monografias em Ciencia da Computacao - PUC-Rio.

1.6

Limitations

As the development of embodied agents is part of a broader context, a set of related aspects are out of the scope of this thesis. Thus, the following topics are not directly addressed by this work:

- Security;
- Ontology;
- Language Description;
- Communication Protocols.

”

2 Background

In this chapter, we first provide an introduction to Multiagent Systems (MASs), including Embodied Agents, which is the domain of this thesis. Next, we define some concepts involved in our proposed solution. In particular, we briefly describe some definitions of (i) modeling formalisms, and some artificial intelligence techniques, such as (ii) neural networks and (iii) evolutionary algorithms. This thesis also involves some concepts that are particular to specific sections, but will not be discussed in this section, such as Internet of Things (IoT) (Section 5.1), Variability Models and Publish-Subscribe (Section 5.3). Thus, we provide a brief description of these additional concepts in the introduction of the correspondent sections.

2.1 Multiagent System

The authors in (Russell and Norvig, 1995) define “**agent**” as “anything that can be viewed as **perceiving** its environment through **sensors and acting** upon that environment through **effectors**.” Accordingly, they describe the sensors and actuators of different agent types. For example, a human agent has eyes, ears, and other organs for sensors, and hands, legs, mouth, and other body parts for effectors. A robotic agent uses cameras and infrared range finders as sensors and various motors as effectors. A software agent has “strings encoded” as its sensors and actuators. They provide an abstract view of an agent, which is illustrated in Figure 2.1. It shows the action output generated by the agent in order to affect its environment.

An agent can inhabit real or simulated environments. These environments can differ taking other properties into account. For example, if the environment changes while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise it is static. Different environment types require different agent in order to deal with them effectively (Russell and Norvig, 1995). Many attributes are discussed in the context of agency, such as (Wooldridge and Jennings, 1995):

- Mobile: an agent that has the ability to move around an electronic

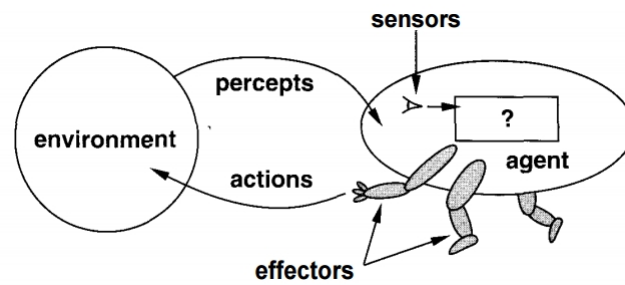


Figure 2.1: The diagram of a generic agent provided by authors in (Russell and Norvig, 1995) [p.32].

network (White, 1994);

- Situated: an agent that experiences the environment using sensors and acts using effectors (Russell and Norvig, 1995);
- Embodied: an agent that has a body and experiences the world directly (Brooks, 1995). According to the author in (Brooks, 1995), disembodied systems concentrate on programming intellectual activities like chess, while the embodied approach aims at equipping a digital computer with the best sense organs (e.g. “organs” to move, talk, hear, touch, and see). We describe these agents in more detail in Section 2.2;
- Awareness: an agent has the ability of sensing its environment in different ways, and take decisions accordingly.

According to the authors in (Wooldridge, 2009), a collection of interacting agents is a Multiagent System (MAS). Therefore, multiagent systems can be used to model complex and dynamic real-world environments, which involves a vast number of entities (e.g. simulation of societies) (Poslad, 2007). It is a useful paradigm for managing large distributed information handling systems (Marzo et al., 2004).

Multiagent systems have been applied to a wide range of application types, including e-commerce, human-computer interfaces, network control, air traffic control, and health diagnosis (Lucena, 2004; Pěchouček and Mařík, 2008).

2.1.1 JADE

The Java Agent DEvelopment Framework (JADE) is a Java software framework implemented that facilitates the development of multiagent systems (Telecom, 2015). According to the authors in (Pěchouček and Mařík, 2008), JADE is a leading open-source agent development environment on the market and some of the existing MAS applications and prototype systems use it.

JADE implements the Foundation for Intelligent Physical Agents (FIPA) specifications that represent a collection of standards for the development of agent-based systems (FIPA, 2015). One of these standards is the Agent Communication Language (ACL) (FIPA, 2015), a protocol for agent communication. It allows the development of an interoperability communication structure, in which agents can be executed on different platforms and exchange information (Bellifemine et al., 2007; Bellifemine et al., 2010).

2.2

Embodied Agents

According to the author in (Brooks, 1995), “only an embodied intelligent agent is fully validated as one that can deal with the real world.” Embodied agents have a body and are physically situated, that is, they are physical agents interacting not only among themselves but also with the physical environment. They can communicate among themselves and also with human users. Robots, wireless devices and ubiquitous computing are examples of embodied agents (Steels, 2004). According to the author in (Steels, 2004), a robot can be seen as a software agent controlling a physical body. For example, the author in (Wooldridge, 2009) describes the robot Stanley (i.e. an unmanned ground vehicle navigation developed by authors in (Thrun et al., 2007)) as “**an autonomous agent embodied in a car.**” The author in (Wooldridge, 2009) states that most effort in building Stanley went into the perception module, which is responsible for making the relation between the sensors and the associated techniques to interpret sensor data. This complexity is not a particular problem of physical agents. The authors in (Russell and Norvig, 1995), for example, support the idea of specifying which action an agent ought to take in response to any given percept sequence to provide a design for an ideal agent. However, they argue that creating these specifications could provide an infinite list for most agents. As depicted in Figure 2.2, the task of making decisions is the most challenging.

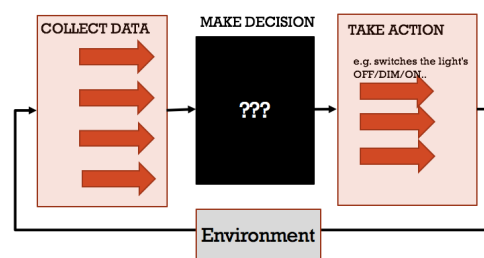


Figure 2.2: The challenge: how does an embodied agent make decisions based on collected data?

To mitigate this problem, some approaches have provided embodied agents with artificial neural networks (Nolfi, 1995). Figure 2.3 depicts a common approach to model physical agents, such as robots and IoT devices.

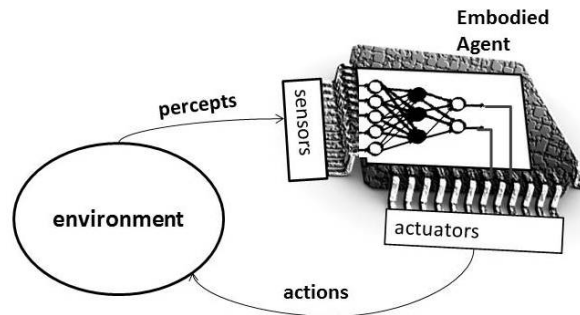


Figure 2.3: A general embodied agent model.

Furthermore, according to the authors in (Polani, 2011), the embodiment can be seen as a two-way filter layer between the controller and the environment. First, the embodiment filters the external world and determines how the controller perceives it. Second, the embodiment translates commands emitted by the controller and expresses them as observable behaviors. Through the Internet, the controller and body can communicate. A software agent contains the controller that is an encoded neural network. The neural network outputs define the values to be set in the actuators. These outputs are calculated according to the data collected by sensors and the encoded configuration that the neural network is using.

2.2.1 Evolving Embodied Agents

Because there is a need to find a configuration that enables those agents to act according to the environment specifications, to find the best configuration for the neural network is not an easy task. Thus, a known approach is to use a genetic algorithm to configure this neural network.

The authors in (Nolfi et al., 2016) describe the process for evolving embodied agents using an evolutionary algorithm, such as genetic algorithm. The interested reader may find more details about this process elsewhere (Miller et al., 1989; Yao, 1999).

Normally, the use of an evolutionary algorithm in a multiagent system provides the emergence of behaviors that were not defined at design-time, such as those in a communication system (Oliveira and Loula, 2015). While in traditional agent-based approaches the desired behaviors are defined intuitively by the designer, in evolutionary ones these are often the result of an adaptation

process that usually involves a larger number of interactions between the agents and the environment (Nolfi and Floreano, 2000).

The process of evolving an embodied agent's neural network can occur on-line or off-line (Nelson et al., 2007). The on-line training uses physical devices during the evolutionary process. In such case, an untrained neural network is loaded into a physical agent. Then, the evolution of this neural network occurs based on the evaluation of how this real device behaves in a specific scenario. The off-line training evolves the neural controller in a simulated agent (Nelson et al., 2007), and then transfers the evolved neural network to a physical agent.

The major disadvantage of executing on-line evolution is the increase in execution time, since evaluating physical devices may require more time. In addition, the training process based on evolution can produce bad configurations for the neural network, which could generate serious problems in particular scenarios. Otherwise, the on-line training ensures that evolved controllers function well in real devices.

2.3

Evolutionary Algorithms

The artificial evolutionary algorithm is briefly defined as a collection of individuals in a search space, where each is a different solution to a given problem. A chromosome represents the individual, and the goal of the search is to identify the one with the best genetic material. We measure the quality (fitness) of each by a given fitness function, which measures how good that particular individual is among the ability of the entire population. The fittest individuals will have greater ability to reproduce and this could result in the reduction of the least fit individuals.

The different individual sequence of genes can be in various formats such as binary, character strings, numeric values, and others. We represent the genetic material of an individual, of that search space, by a vector m with p positions:

$p = x_1, x_2, x_3, \dots, x_m$, where each x represents a gene also known as solution variable.

According to the authors in (Floreano and Mattiussi, 2008), the size of the population for experiments that require interaction with a real environment normally is less than one hundred. Typically, the genotype of the first individuals are created randomly. The goal of this process is to ensure diversity for the initial generation. In the case of binary representation, for example, each genotype is created by a random sequence of 1s and 0s. The quality of the generation is measured based on an individual fitness average.

While the algorithm does not find a generation to meet the stopping criterion, new generations are being formed. The stopping criterion can be an individual performance criteria, by fitness function, the average fitness of all individuals, or the range of the maximum number of generations formed during the evolutionary process.

Natural selection preserves the best-adapted individuals of a generation, giving them a greater chance to procreate. Following this concept, the algorithm should select the best parents of the current generation, so that they can transfer their genetic material to the next generation. Thus, the first step of the reproduction process is the selection. There are several ways to perform this selection process. Among the best known are the roulette, selection by a tournament, and by ranking (see (Floreano and Mattiussi, 2008)).

To avoid the loss of the best solution during evolution, one can use the popular strategy of elitism. Elitism is a strategy of a composition of a new population. The strategy maintains the n best-selected individuals in the new generation, so that, one of their children has a copy of their genetic material.

When a generation does not meet the stopping criteria, among all possible solutions to the problem it addresses, probably the best of their solutions is not enough to solve it. Thus, it is necessary to apply genetic operators, so that it is possible to introduce diversity in the population, and thus can insert new solutions in that search space. One of the operators is the crossover, which generates a new individual from the combination of genetic material from two or more individuals parents. The other is the mutation that generates new individuals with a slightly changed genetic code from small random changes caused by their genotype. For more details about the different types of selection and genetic operators, see (Floreano and Mattiussi, 2008).

2.4

Artificial Neural Network (ANN)

Bodies with a neural network have at least two advantages: selective transmission of signals among body parts (input-output mapping (Haykin, 1994)), and adaptation through synaptic plasticity (Floreano and Mattiussi, 2008). The block diagram in Figure 2.4 illustrates the input-output mapping process of a nervous system. It receives information from the environment, perceives it, and makes appropriate decisions (Haykin, 1994). According to (Haykin, 1994), “plasticity permits the developing nervous system to adapt to its surrounding environment.” As a result, the interest of modeling the operation of the neural system to build intelligent machines has been increasing.

Haykin (Haykin, 1994) provides the following definition of a neural

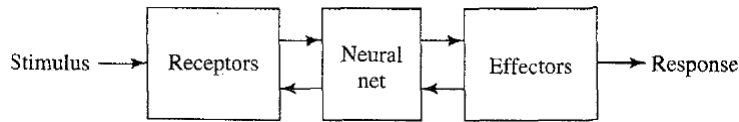


Figure 2.4: Block diagram representation of nervous system (Haykin, 1994). Pg.28.

network viewed as an adaptive machine:

A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the controller in two respects:

- Knowledge is acquired by the network from its environment through a learning process.
- Inter-neuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

Systems that use an Artificial Neural Networks (ANN) have been presenting other properties beyond adaptivity, such as pattern recognition, perception, and motor control (Haykin, 1994). An ANN consists of neurons and connections. To understand how a neural network functions is necessary to understand these elements.

2.4.1 Artificial Neuron

According to Haykin (Haykin, 1994), a neuron is an information-processing unit that is fundamental to the operation of a neural network. Figure 2.5 shows the model of a neuron, which forms the basis for designing artificial neural networks.

This model identifies some basic properties of a neuron, as shown (Haykin, 1994):

- A set of **synapses** or connecting links. Specifically, a signal x_m at the input of synapse m connected to neuron k is multiplied by the synaptic weight $W_k m$. Unlike a synapse in the controller, the synaptic weight of an artificial neuron may lie in a range that includes negative as well as positive values.
- An **adder** for summing the input signals, weighted by the respective synapses of the neuron. The adder can add excitatory inputs or subtract

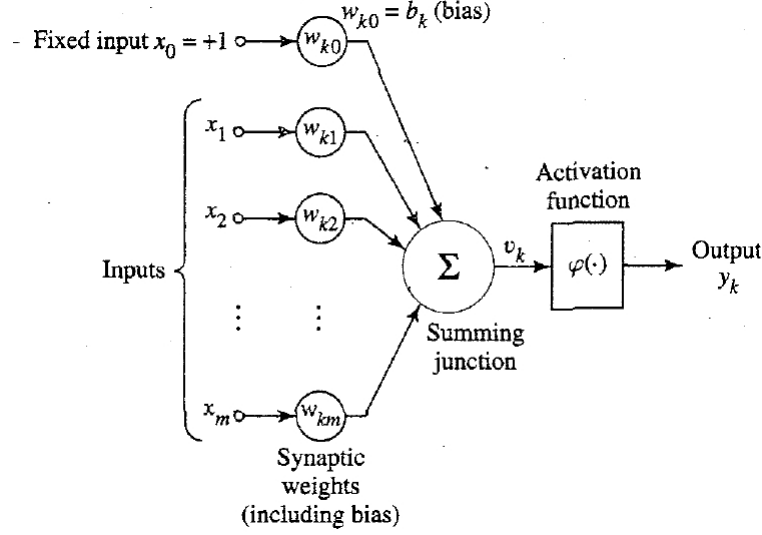


Figure 2.5: The model of a neuron (Haykin, 1994). P. 33.

inhibitory inputs from other neurons connection (McCulloch and Pitts, 1943).

- An **activation function** for limiting the amplitude of the output of a neuron. Typically, the amplitude range of the output of a neuron is normalized and written as a unit closed interval $[0,1]$ or alternatively $[-1,1]$.
- An externally applied **bias**, denoted by b_k . The bias b_k has the effect of increasing or lowering the net input of the activation function, depending on whether it is positive or negative, respectively. As shown in Figure 2.5, the effect of the bias is accounted by adding a new input signal fixed at $+1$, and adding a new synaptic weight equal to the bias b_k .

In mathematical terms, Haykin (1994, P. 33) proposes the following equations to describe a neuron k :

$$u_k = \sum_{j=1}^m w_{kj} \times x_j \quad (2-1)$$

$$v_k = u_k + b_k; \quad (2-2)$$

$$y_k = f(v_k); \quad (2-3)$$

where x_1, x_2, \dots, x_m are the input signals; $w_{k1}, w_{k2}, \dots, w_{km}$ are the synaptic weights of neuron k ; u_k is the linear combiner output due to the input signals; b_k is the bias; $f(\cdot)$ is the activation function; and y_k is the output signal of the neuron.

The activation function, denoted by $f(\cdot)$, defines the output y of a neuron k in terms of the induced local field v . According to Haykin (1994), the sigmoid

function is by far the most common form of activation function used in the construction of artificial neural networks. The equation 2-4 presents an example of a sigmoid function:

$$y_k = \frac{1}{(1 + e^{-v_k})}; \quad (2-4)$$

See Haykin (1994, P. 34) for more information about different types of activation functions.

2.4.2

Network Architectures

An ANN is an arrangement of neurons. There are different classes of network architectures (e.g. competitive networks, which can make use of a Winner-Takes-All algorithm(WTA), recurrent, feedforward, etc. See a discussion about them in (Haykin, 1994)). A widely used type is the feedforward network. They contain an input layer and an output layer. The first consists of sensory neurons that receive environmental stimuli. The second consists of motor neurons, which are responsible for producing the network response (Flo-reano and Mattiussi, 2008). A network may have one or more hidden layers, which are composed of hidden neurons.

The function of hidden neurons is to intervene between the external input and the network output in some useful manner (Haykin, 1994). These neural networks are commonly referred to as multilayer perceptron (Haykin, 1994). To determine the number and the size of the hidden layers is mostly a matter of trial and error. However, there are heuristic techniques to establish an optimal number of hidden neurons (Haykin, 1994). For example, the author in (Kuorkova, 1992) proposes a technique to derive estimates of numbers of hidden units based on properties of the function being approximated and the accuracy of its approximation. Alternatively, some researches (Miller et al., 1989; Belew et al., 1990) have been using evolutionary algorithms to design the network topology automatically.

Another type of network class is the recurrent network. Recurrent networks distinguish themselves from feedforward networks in that they have at least one feedback loop. Feedback refers to a dynamic system whenever the output of an element in the system influences in part the input applied to that particular element (Haykin, 1994).

2.4.3

Adaptive Process

According to Floreano and Mattiussi (2008), adaptation is a major feature of the nervous system. This allows the body to modify and develop behaviors in order to maintain or improve their probability to survive in dynamic and partially unknown environments. According to the authors in (Yao, 1999), learning and evolution are two fundamental forms of adaptation. There has been a great interest in combining learning and evolution with artificial neural networks (ANNs) in recent years.

Learning Algorithm

According to Haykin (1995, P. 46), “a major task for a neural network is to learn a model of the environment in which it is embedded.” The learning algorithm used to train the network is directly linked with the neural network structure. After the learning step, the knowledge representation of the surrounding environment is defined by the values taken on by the free parameters (i.e., synaptic weights and biases) of the network (Haykin, 1994).

A learning algorithm can be supervised or unsupervised. In supervised algorithms, the neural training process is performed using labeled examples. In such cases, each example representing an input signal is paired with a corresponding desired response. Algorithms for unsupervised learning do not provide a set of input-output pairs (Haykin, 1994). According to the authors in (Floreano and Mattiussi, 2008), the use of an evolutionary algorithm is an alternative or complementary technique to unsupervised learning algorithms for adapting a neural network. We describe a particular evolutionary algorithm in Section 2.4.3.

An algorithm that is commonly used to train multilayer perceptrons is the error back-propagation algorithm (Haykin, 1994). Basically, error back-propagation learning consists of two steps: a forward and a backward steps. In the first one, an activity pattern (input vector) is applied to the sensory nodes of the network. As a result, a set of outputs is produced. During the forward step the synaptic weights of the networks are all fixed. During the backward step, the synaptic weights are all adjusted in accordance with an error-correction rule (Haykin, 1994). The adjustment of the synaptic weights of the neurons in accordance with the error signal leads to an adaptive process (Haykin, 1994). See a detailed description about back-propagation algorithm in Haykin (1994, P. 183).

Evolving Neural Networks with a Genetic Algorithm

Similar to unsupervised learning algorithms, evolutionary algorithms have been commonly used for adapting neural networks without a teacher (Turner and Miller, 2014). According to Pagliuca et. al (Pagliuca et al., 2018), neuroevolution (NE) can be applied to any type of neural network and can be used to adapt all the characteristics of the network, including the architecture of the network, the transfer function of the neurons, and the characteristics of the system (if any) in which the network is embedded.

For the evolution of a neural network, its characteristics are encoded in artificial genomes. A genome is usually represented as a string of real or binary values, and evolved according to a performance criterion. If the goal is only to train the neural network, the genotype will encode only the value of synaptic weights (Floreano and Mattiussi, 2008). The interested reader may consult a more extensive paper (Yao, 1999).

By using an evolutionary algorithm, a weight sequence represents the genotype of an individual. One generation consists of a pool of individuals that represent different network configurations (see the description of evolutionary algorithms in Section 2.3). Figure 2.6 illustrates a multilayer feedforward network and two candidates (individuals) for its weights sequence. We are supposing that the weights are real values and can be written as a unit closed interval $[-3,3]$.

For each weight sequence candidate, the algorithm evaluates the network performance. The better individuals are selected to reproduce and create the next generation.

2.5

Formal Methods

Formal specification is the process of using a formal method, a language with a mathematically defined syntax and semantics, to describe a system (Clarke and Wing, 1996). According to Clarke et al. (1996) (Clarke and Wing, 1996), the main benefit of using a formal method is to gain a deeper understanding of the system that has being specified. It allows developers to uncover design flaws, inconsistencies, ambiguities, and incompletenesses.

There are many formal methods available, such as Z notation (Spivey, 1988), Temporal Logic (Wolper, 1983), and Statecharts (Harel, 1987). According to (Ingrand, 2019), we must select a formal method according to the type of behavior that we have to model, such as functional and timing behavior, and the type of properties that we want to check. For example, the Z notation

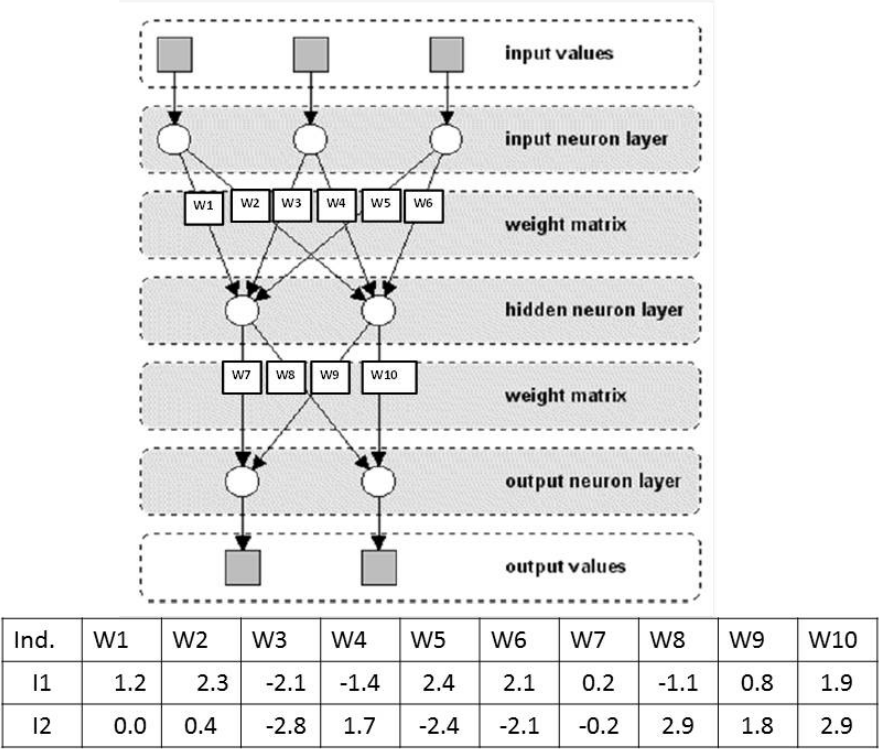


Figure 2.6: Evolving a neural network. Adapted from Floreano and Mattiussi (2008, P. 317).

is usually used to specify the behavior of sequential systems, while Statecharts are usually used to specify the behavior of concurrent systems.

2.5.1
Statecharts

Statecharts is a formal method that extends the formalism of state machines and state diagrams with essentially three elements: hierarchy, concurrency and communication (Harel, 1987). This formal method has been used to specify reactive systems, that are systems that have to react to external and internal stimuli. Accordingly, the behavior of reactive systems is composed of input and output events, conditions, and actions (Harel, 1987). The main elements of Statecharts are:

- State and Events: a natural description of the dynamic behavior of a complex system (Harel, 1987);
- Transition: a transition is the connection between two states (source and destination) and can be represented by the triple: $e[c] / a$, in which “e” is an event, “c” is a condition, and “a” is an action. The transition “ $e[c] / a$ ” is enabled if the statechart is in a source state for this transition, the event “e” occurs, and the condition “c” is true. Thus, the trigger of the

transition is the event and condition together. If a transition is applied, the state changes from the source to the destination state, and the action is carried out;

- Composition (or Clustering): introduces the XOR (exclusive-or) decomposition of states, which captures the property that, being in a state, the system must be in only one of its composite components;
- Orthogonality: independence and concurrency. It introduces the AND decomposition of states, which captures the property that, being in a state, the system must be in all of its AND components. According to Harel, “an obvious application of orthogonality is in splitting a state in accordance with its physical subsystems. This typically occurs on a very high level of the specification”;
- History. In statecharts syntax, entering the history state means to enter the most recently visited state.

In short, state diagrams are simply directed graphs, with nodes denoting states, and arrows denoting transitions. Depending on the tool that is used to design statecharts, a state can be represented by different graphical notations. States are usually represented by rounded rectangles. Figure 2.7 depicts an example that we reproduced from Harel (1987, P. 14) using the StarUML tool (StarUML, 2019).

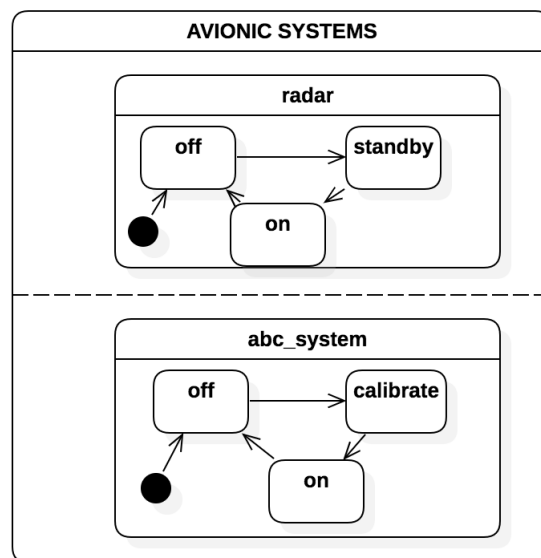


Figure 2.7: Example of orthogonal components. Adapted from Harel (1987, p.14).

This example depicts the statechart of an overly simplified avionic systems, composed of a set of subsystems such as radar and abc-system. The notation used in statecharts is the physical splitting of a box into

components using dashed lines. According to Harel, to be useful, a state approach must be modular, hierarchical and well-structured. The possibility of modular modelling also provides easy assembly and reusability. Another characteristic that we represent in this picture is the orthogonality. As shown, the avionic subsystems are orthogonal components, that means being in the “AVIONIC SYSTEMS”, the system must be in “radar” and “abc-system”.

In Chapter 4, we provide more details about statecharts, following Harel’s formal description (Harel, 1987).

3

Related Work

As described in Chapter 1, there are many challenges related to understanding the relationship between the body and the controller of an embodied agent because of the complex dynamic interactions between the agents and the environment. According to (Harel et al., 2019), approaches that integrate Artificial Intelligence techniques, such as Machine Learning and Multiagent Systems, and Software Engineering (SE) Techniques are promising. SE techniques, such as statecharts, can endow AI-based systems with intuitive and clear specifications that can result in systems that are much easier to enhance and maintain compared to current uses of AI in system development.

In this chapter, we describe current research efforts to specify embedded agents and reconfigurable systems. In addition, we introduce some approaches and applications that support or involve the development of self-configurable IoT embodied agents.

3.1

Reference Models for Agents

The need of providing reference models to understand the relationships among the properties of agents, environments, and forms of interaction between them has been discussed in agent-based literature for many years (Agre, 1995). According to (Agre, 1995), the characterization of interaction should allow us to address questions like these:

- What will our agent do in a given environment?
- Under what conditions will it achieve its goals or maintain desired relationships with other things?
- In what kinds of environment will it work?
- How do particular aspects of an environment affect particular types of agents' abilities to engage in interactions that have particular properties?
- What forms of interaction require an agent to employ particular elements of internal architecture, such as memory?

- What forms of interaction permit an agent to learn particular knowledge or skills?

According to (Klügl and Davidsson, 2013) and (Santos et al., 2017), there is clearly a lack of formal specification in Multi-Agent Based Modelling and Simulation (MABS). Wooldridge (Wooldridge, 2009) presents formal descriptions for the basic introduction to multiagent systems, such as what is an agent and how it is embedded in an environment. A more elaborated formal description for multiagent systems can be found in (Luck et al., 1995; d’Inverno et al., 2004). Luck et al. (1995) propose a three-tiered hierarchy of entities comprising objects, agents, and autonomous agents. Based on this hierarchy, they present a formal approach for agents using Z notation. Luck et al. (1995) describe objects as entities that are capable of perceiving the environment and acting. Accordingly, actions are events that change the state of the environment, and attributes are perceivable characteristics of this environment. The authors argue that agents are objects that serve useful purposes, satisfying a goal or a set of goals. In addition, an agent may not necessarily be able to perceive the environment.

The formalization for agents proposed by Luck et al. (1995) does not consider changes that may suddenly happen in the environment by perturbing the agent’s state, a situation that is usual to entities structurally coupled with their environment, as embodied agents. Notwithstanding, Bae and Moon (2015) (Bae and Moon, 2015) investigated a set of specification proposals for agent-based models and observed that most of them focus more on agent behaviors than on the environment. Our proposal considers that the environment has autonomous features, which can be changed by generating perturbations to the agent.

In addition to the coupling with a dynamic environment, embodied agents also have other particularities, such as many options to configure their physical properties, and the use of an artificial neural network to make decisions. Considering that embodied agents is a subtype of autonomous agents, as described in (Franklin, 1997), we could have extended an existing reference model to include embodied agent particularities. For example, the one provided by Luck et al. (1995), that still describes autonomous agents. However, to extend an existing reference model to include embodied agent particularities, a set of elements from the Agent Oriented Software Engineering (AOSE) literature that are not usual in the embodied agent literature, such as plans, beliefs and motivation, should be considered.

3.1.1

Reference Models for Embodied Agents

Embodied agents is a subtype of autonomous agents (Franklin, 1997). (Ingrand, 2019) investigates the availability of formal models in autonomous physical systems (ASs) and states that there exist numerous frameworks to develop and deploy AS and robotic software, but most of them do not provide formal models to describe what they do. According to Ingrand (2019), this poses some new challenges for the development of robust and safe ASs that have not yet been addressed by non ASs, since it makes it more difficult to validate and verify ASs. For example, the authors of (Seshia et al., 2016) list five challenges for designing verified AI-based systems: environment modeling, modeling system that learn, formal specification of the desired properties of the system, computational engines for training and testing AI-based systems, and designing a training process that leverages the specification and environment models. According to Ingrand (2019), researchers are just starting to look at these issues.

A research that influenced our proposed reference model is the meta-model proposed in (Klügl and Davidsson, 2013). Klügl and Davidsson (2013) propose an abstract meta-model as a general formalization of multiagent models that considers environmental dynamics that happen without being triggered by an agent, such as seasonal temperature, a tree growing, or rain starts to fall. Also, their proposal is the first formalization for MAS to introduce the concepts of body and controller. In short, an agent consists of both body and controller, in which the body represents the physical parts of the agent by carrying sensors and actuators, and the controller contains the reasoning capabilities by handling the decision-making processes. For example, in the case of Belief–Desire–Intention agents (BDI), the authors state that the controller contains beliefs, goals and plans. The authors also mention that this controller may be based on a neural network, but they only provide a very short explanation about how their model can represent a neural network. Beyond the Body and Controller components, their model also includes the Region component, which represents the environment where the agents and objects are situated. However, as the authors state, their model is very simple and its current state covers only very basic structures and processes. In addition, it is clear the need of addressing an extension of the controller and of providing more explanations about the interactions among the components. For example, it is not clear how the body affects an agent’s controller, neither how the agent’s controller interferes on its behavior. In addition, their model does not encompass support for agent reconfiguration.

3.2

Reference Models for Reconfigurable Systems

The authors in (Luckcuck et al., 2018) systematically survey the state-of-the-art in formal specification for autonomous robotic systems. According to the authors, specifying reconfigurable autonomous physical systems is a challenge.

According to (Bruni et al., 2015), adaptive and reconfigurable systems address other questions, such as :

- What parts of the system should be adapted? That is, which artifacts (variables, components, connectors, interfaces, etc.) have to be modified in order to adapt?
- Why is adaptation required? Is the purpose of adaptation to meet some robustness criteria, to improve the system's performance or to satisfy some other goal?
- How should adaptation be applied? That is, which is the plan that establishes the order in which to apply the necessary adaptation actions?
- Who should enact the adaptation? Which entity (e.g., human controller, autonomic manager) is in charge of each adaptation?

(Weyns et al., 2012) describe a formal reference Z model for self-adaptive systems, called FORMS. The authors state that while FORMS offers a formally founded vocabulary for the key architectural constructs comprising self-adaptive systems, it does not provide an implementation framework from which self-adaptive applications can be derived. Our approach includes some aspects of their reference proposal, such as how the physical agent interacts with the environment.

Another influence on the work presented herein is the architecture for reconfigurable embedded systems proposed by (Karsai and Sztipanovits, 1999). They use a finite-state machine (FSM) to represent reconfiguration, setting a state for each possible configuration. In their architecture, they consider an evaluator component that is responsible for controlling the transitions among the configuration states. This component is also responsible for triggering reconfiguration. In our model, we do not represent alternative configurations, since to pre-define possible configurations to applications based on embodied agents is not a suitable solution, as we discuss in Section 5.2.1. However, we consider an evaluator component that is able to trigger and manage reconfiguration.

3.3

Approaches and Applications for Embodied Agents

There are known software approaches to assist with the development of embodied agents. For example, the Framework for Autonomous Robotics Simulation and Analysis (FARSA) (Massera et al., 2013) assists research in the area of embodied cognition, adaptive behaviour, language and action. A set of works on embodied agents (Marocco and Nolfi, 2007; Nolfi and Parisi, 1997; Nolfi and Floreano, 1998; Massera et al., 2014) was developed using FARSA or related software. Most of these works present a group of embodied agents that evolves for the ability to solve a collective problem.

Another framework for embodied agents is presented in (Sobe et al., 2012), the Framework for Evolutionary Design (FREVO). The authors in (Sobe et al., 2012) present Frevo as a multi-agent tool for evolving and evaluating simulated systems. The authors state that Frevo allows a framework user to select a target problem evaluation, controller representation and an optimization method. This framework is often applied in the creation of robotic applications, such as foraging and navigation tasks.

Our work is the first to introduce embodied agents in the Internet of Things domain, showing that a concept that was very oriented towards the development of robots, could be extended to a new application domain (Nascimento and Lucena, 2017). Our work was extended by (Zedadra et al., 2017), which provided a reference architecture for the development of Internet of Things applications based on cognitive physical agents. According to the authors, the use of a swarm of simple devices provides scalability, robustness, flexibility and interoperability to IoT architectures. As future work, they intend to implement the proposed reference model and to investigate its application to different IoT systems.

Both reconfigurable-based and physical agent applications have been successfully applied in several scenarios, such as in the health-care, transportation, military and smart city domains. Reconfigurable systems are usually associated with the provision of customized recommendation or decision making. According to (Macías-Escrivá et al., 2013), most of these applications are only prototypes and are not in production because users may not understand or trust them.

Embodied agents has been used to robotic applications so far. However, there are many examples of applications that could be modeled by using the concept of embodied agents. As an example, the smart paper machine presented in (Katasonov et al., 2008), where the machine has a set of sensors to self-monitor, and activates an alarm if it needs maintenance. The recon-

figuration process occurs through changing the formats of messages or the algorithms for issuing alarms. The authors in (Baresi et al., 2014) simulate a smart green house scenario with intelligent rooms. In their simulation, flowers are distributed in different rooms based on specific characteristics. If a flower is sick, it will be allocated to another room or the room's configuration will change. Therefore, the authors use adaptive techniques to perform discovery, self-configuration, and communication among heterogeneous things. The authors in (Zhu et al., 2014) simulate the development of a smart office, which is composed of people, rooms and resources, such as air conditions, networks, lighting, and temperature. The goal of their proposal is to optimize an office, by managing tasks and the organization. The use of an adaptive algorithm allows the system to infer the process when the environment changes. If a person leaves a room, the system recalculates the office organization. According to the authors, "the adaptations are driven by a declarative algorithm, which considers the organization and the task."

The website in (Postcapes, 2019) lists more than fifty smart applications for the Internet of Things. An example is a smart garden. The idea of this project is to allow people to get recommendations of plants that will grow well. The adaptation occurs because the system can be adjusted according to users' experiences via a shared database. Another example is a smart landfill gas collection. The system acts independently and can be adjusted according to an operators' need, whether that be maximizing collection, minimizing odors, or collecting gas of a specific chemical composition. However, as these applications are listed in a website as commercial products, the authors do not provide information about their development (e.g. models, techniques).

4

Fundamentals of Reconfigurable Embodied Agents

In this chapter, we cover the question **RQ1. How can embodied agents and their interactions with the environment be specified?**. This work proposes a reference model that offers some well-suited abstractions tailored to the development of self-configurable IoT embodied agents that are able to interact with dynamic environments. First, we survey the main concepts related to embodied agents and their requirements, taking previously published work into account. Second, we provide high-level statecharts models of embodied agents, and then we expand the boxes of each component (e.g., Body, Controller). After describing general models, we extend these models to represent an illustrative application that uses reconfigurable embodied agents.

4.1

Preliminary embodied agent concepts

Embodied agents are “autonomous agents structurally coupled with their environment” (Franklin, 1997). According to (Quick et al., 1999), a system X is embodied in an environment E if perturbatory channels exist between the two. In other words, X is embodied in E if for every time t at which both X and E exist, some subset of E ’s possible states have the capacity to perturb X ’s state, and some subset of X ’s possible states have the capacity to perturb E ’s state (Quick et al., 1999). **Accordingly, our proposed model must contain at least two components: embodied agents and the environment.**

4.1.1

Agent Body

Basically, an agent body is composed of sensors and effectors. According to (Kinny, 2001), the agent “interacts with its environment via interfaces of two types: sensors from which the agent receives events that carry information from its environment, and effectors by which the agent performs actions that are intended to affect its environment.”

Auerbach and Bongard state that “different parts of the robot’s body are responsible for different behaviors. For example, wheels or legs may allow for

movement while a separate gripper allows for object manipulation” (Auerbach and Bongard, 2009). **Based on the previous observations, our model describes the effects that the agent body has on its behavior.**

4.1.2

Agent Behavior

To develop autonomous agents, (Horn, 2001) propose the development of agents that execute a control loop composed of four activities: collect, analyze, plan and execute, which are briefly described next.

- **Collect:** collect application data;
- **Analyze:** analyze the collected data by trying to detect problems;
- **Plan:** decide on an action in the case of problems; and
- **Execute:** change the application because of executed actions.

Based on the initial concepts of embodied agents, we customized this control loop proposed by (Horn, 2001) to define the behaviors of embodied agents. Instead of executing the analyze and plan activities, embodied agents make decisions based on a controller, which could be a finite state machine (FSM) or an artificial neural network (ANN), as shown in Figure 4.1.

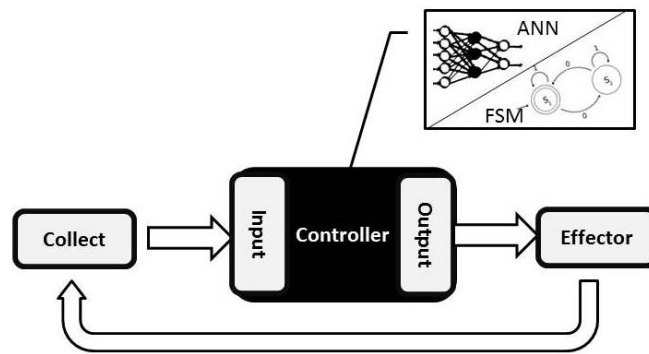


Figure 4.1: Control loop executed by embodied agents.

In short, an embodied agent must execute three key activities in sequence, namely: (i) collect data from the thing; (ii) make decisions; and (iii) take actions. The task of data collection focuses on processing information coming from devices, such as reading data from input sensors. The collected data are used to set the inputs of the agent’s controller. Then, the controller processes a decision to be taken by the agent. Embodied agents act based on the controller output. An action (effector activity) can be to interact with other agents, to send messages, or to set actuator devices, thus making changes to the environment.

4.1.3

Agent Controller

In addition, the intelligent behavior of an embodied agent arises out of the coupled dynamics of its body, its controller, and the environment where it is situated (Auerbach and Bongard, 2012). According to (Auerbach and Bongard, 2009), the complexity of an agent controller and body must match the complexity of a given task. However, more complex task environments require the agent to exhibit different behaviors. Therefore, it is necessary to find the combination of agent body and controller that allows the agent to behave accordingly.

Based on the previous observations, our model also describes the effects of the agent's controller on its behavior. In addition, it includes a new component to represent task environment.

4.2

Preliminary reconfiguration concept

According to (Karsai and Sztiapanovits, 1999), the reconfiguration mechanism must be associated to an evaluation module that represents how the system's performance will be monitored and evaluated, and how the evaluation's result will affect the system's architecture. In addition, the reconfiguration mechanism should be capable of interacting with the evaluator, being triggered by it.

The evaluation module measures a set of variables from the environment, and it can trigger the reconfiguration mechanism if it identifies a change in the set of measurable variables.

Based on the previous observations, our model includes a new component to represent task evaluation, which is able to trigger the reconfiguration process.

4.3

Statecharts

Statecharts makes it possible to view the description of the solution in different levels of detail (zooming-in and zooming-out), allowing a top-down behavioral specification. Therefore, it facilitates the understanding of a complex architecture. Other relevant characteristics are compositionality, hierarchy, and inter-components communications. Thus, our model can use transitions to explain the perturbations that occur between the environment and agents, which are the main components of embodied agents, as explained in Section 4.1.

A high-level specification is given in Figure 4.2, which contains the full statechart of an embodied agent. The main components of configurable embodied agents will be described in detail in the next subsections, in which we will look inside each one of these components.

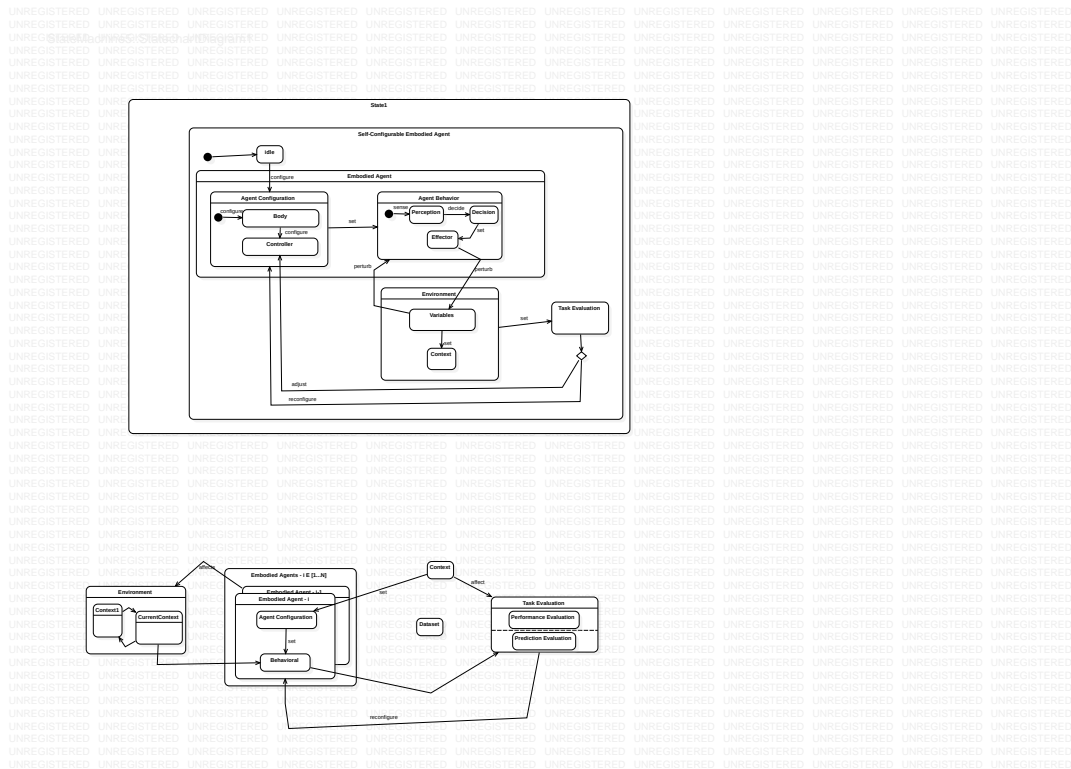


Figure 4.2: General statechart of embodied agents - putting the main components together.

The diagram above shows the states that are responsible for configuring embodied agents according to their task. The agent comes out of the idle state when the system is initialized. As shown, we represented an embodied agent as a super state, and we assumed some physical and functional description of the system to describe their actions and activities. Therefore, we provided an hierarchical decomposition of this state into agent configuration and agent behavior. The component of agent configuration is responsible for configuring the agent's body and controller. Once the agent is configured, it is ready to behave in accordance with the environment. The agent behavior consists of three components: i) perception; ii) decision; and iii) effector. As shown, the perception, decision and effector capabilities of the agent are directly related to how the body and controller of the agent are configured. To represent it, we use the transition "set" between the components of configuration and behavior. Internal transitions have been omitted for simplicity.

Figure 4.2 also depicts inter-component communications between the agent behavior and environment components. As shown, the variables of the

environment perturb the agent behavior. As Perception is the default state among Perception, Decision and Effector, the default way of entering this group of states is by the Perception state. In short, environment's variables perturb agent's perception. On the other hand, the effector state perturbs the environment. We will describe the details of these interactions in subsequent sections.

The unique way of entering the Task Evaluation component is through the environment component via “set.” As shown in Figure 4.2, after evaluating the task, the system return to the initial state, that is the agent configuration, resulting on a cycle. By entering the agent configuration state, the default option is to reconfigure the whole agent unless transition “adjust” is selected. If “adjust” is selected, only the agent controller will be reconfigured.

4.3.1

Agent Configuration - Body and Controller

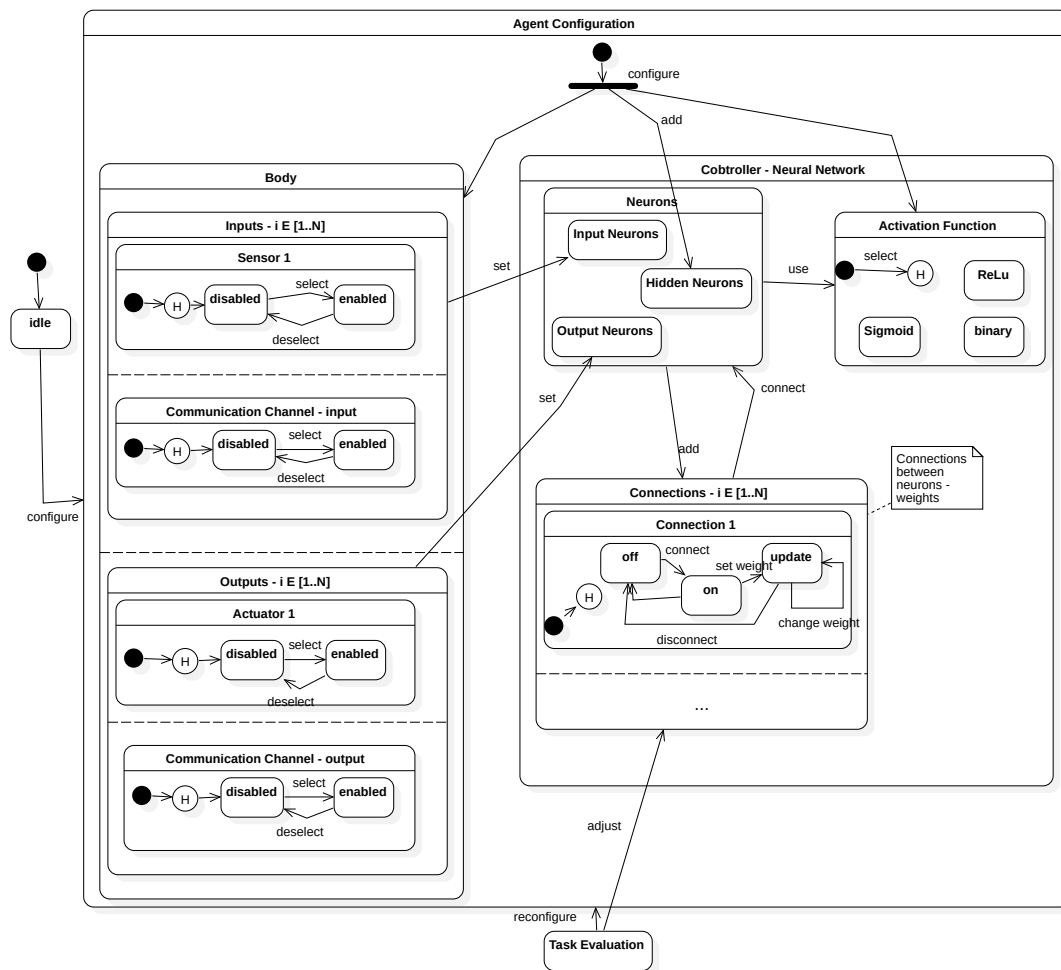


Figure 4.3: Body and Controller Configuration components of the embodied agent statechart.

A refinement of the agent configuration state yields Figure 4.3. After looking inside the body component, we can see that the process of configuring agent body consists of enabling or disabling some inputs, such as sensors, and outputs, such as actuators, which are represented with disabled/enabled substates. According to Harel (1987) (Harel, 1987), “an obvious application of orthogonality is in splitting a state in accordance with its physical subsystems.” Thus, we represent the process of configuring agent body’s inputs and outputs as independent states. In other words, selecting the components to compose the body’s outputs does not depend on the components that are selected to compose its inputs. By default, all inputs and outputs are unselected, but we include a shallow history (“H”) state in each component to allow the system to enter the most recently visited of the two, and enter “disabled” if the system is there for the first time.

To configure the agent controller, there is a transition between the body’s inputs and the input neurons, and a transition between body’s outputs and output neurons. In other words, input neurons are set according to the enabled body’s inputs, and the output neurons are set according to the enabled body’s outputs. Each neuron may be connected to more than one neuron. So, a neuron adds a connection to the system and this connection is connected to a neuron (another or the same one). This statechart component represents the weight configuration of a neural network, in which the output of a neuron may enter another neuron as an input with an specific weight. After zooming-in the agent’s controller, it is possible to see that the “adjust” transition segment, whose source is a state at the Task Evaluation component, is directly associated to the connections component, allowing the reconfiguration process to result only on enabling, disabling or updating connections.

4.3.2 Agent Behavior

As we explained above, the behavior of the agent varies based on the physical components that are operated by the agent and its controller. In short, the behavior of embodied agents is composed of three activities (see subsection 4.1.2): perception, decision and effector. The task of perception focuses on processing information coming from devices, such as reading data from input sensors. The collected data are used to set the inputs of the agent controller. Then, its controller processes a decision to be taken by the agent. Finally, the agent acts based on the controller output. An action (effector activity) can be to interact with other agents, to send messages, or to set actuator devices, thus making changes to the environment.

Selected inputs have an effect on the perception state, while outputs determine the effector state. For example, if the communication input is enabled, the agent will be able to listen to other agents; if the agent has a sensor A, the agent will be able to sense variable A at the environment; and if the communication output is activated, the agent will be able to speak.

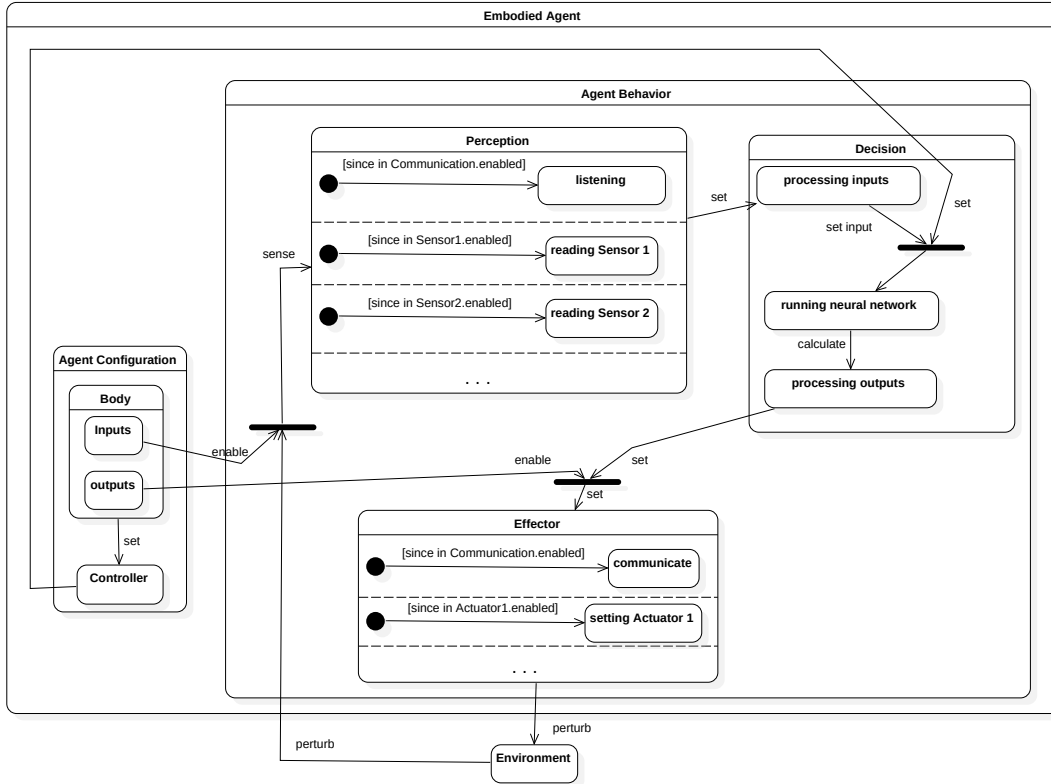


Figure 4.4: Behavior component of the embodied agent statechart.

Note that we have used some joint states on this picture. When a state emanates from a joint state, its source set consists of the sources of those of the constituent segments. For example, entering the substate “running neural network” depends on the incoming transitions from the states “configuring controller” and “processing inputs”. In other words, how the collected inputs are processed to produce outputs depends on the neural network configuration. Consequently, the controller is crucial to the decision state. Perception and Effector substates also have more than one source. The transition segment “perturb” from Environment and the segments “enable” from Inputs are connected to enter the Perception’s substates, and this means that sensing a specific environment’s variable depends on the sensors that the agent has. Upon sensing some sensors on “Perception” component, the “set” transition will be taken and the action “processing inputs” will be carried out. Notwithstanding, entering Effector’s substates depends on the outgoing transitions from body’s outputs (e.g. actuators that are enabled) and from the Decision state (e.g.

output values that were calculated by the neural network). Results from Effector component perturb the environment, since some variables can be updated according to the agents' actions. For example, if the agent turns on the light, it will change the value of the brightness variable.

4.3.3 Environment and Task Evaluation

The “Environment” component is composed of variables and contexts, as depicted in Figure 4.5. Variables represent perceivable characteristics of this environment. The “Variables” component consists of N orthogonal substates, in which each substate is responsible for controlling an specific variable of the environment, such as brightness, humidity, and many others. They are actually responsible for the updating itself. Based on the variables' values, an specific context is selected. For example, supposing we have the contexts of “day” and “night”, the current context will be selected according to the values of “brightness” and “time” variables.

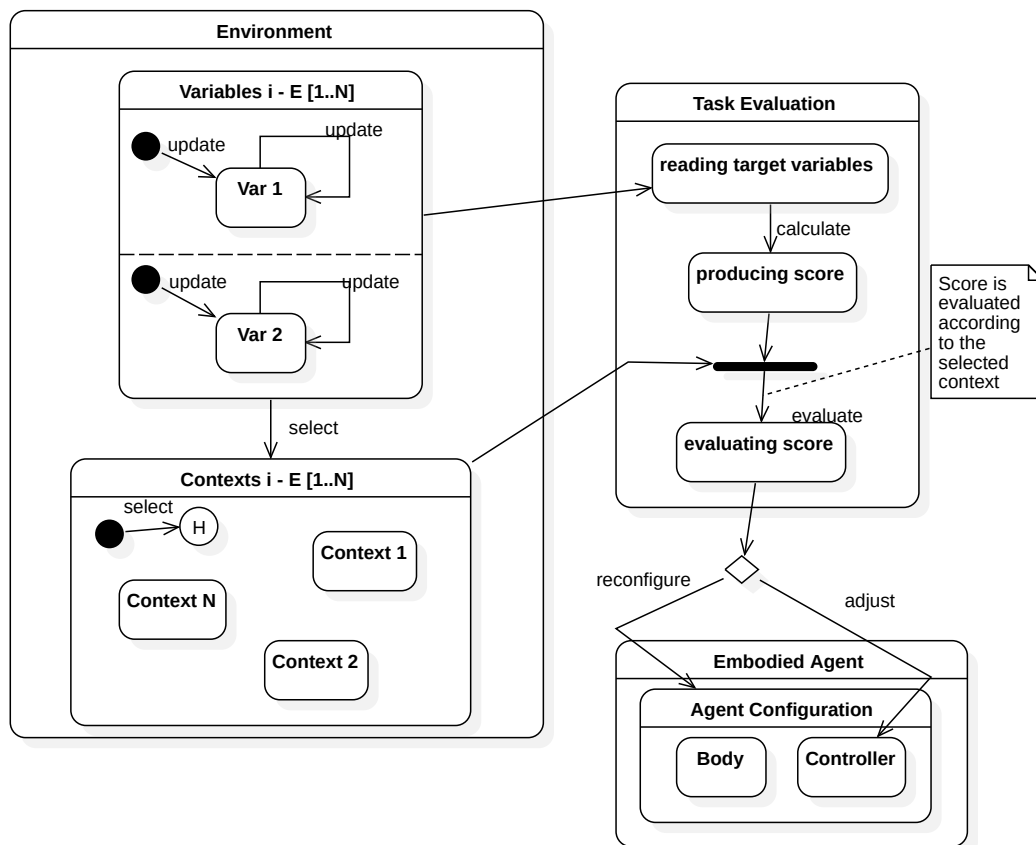


Figure 4.5: Statechart of Environment and Task Evaluation components.

In short, the “Task Evaluation” component is responsible for examining the environment to investigate how the collection of embodied agents can be

configured to deal with the system’s requirements and environmental changes. For this purpose, it will inspect specific variables on the environment in order to calculate a score. However, the significance of this score varies according to the current context, as we discuss in (Nascimento et al., 2018). In (Pezzulo and Nolfi, 2019), the authors illustrate a scenario with artificial agents that the evaluation policy varies according to the context. We represent this situation with a joint state. For example, if we are calculating energy consumption, it is expected that the energy that is spent during the night be greater than during the day. So consuming 10kW during the day has a different impact from consuming this same energy value during the night. Further, the agents reconfiguration will be operated according to this evaluation. It can trigger the transition “reconfigure” by restarting the whole process of configuring an agent, as shown in Figure 4.2, or just trigger the transition “adjust”, reconfiguring only the neural network connections, as shown in Figure 4.3.

4.3.4 Illustrative Application

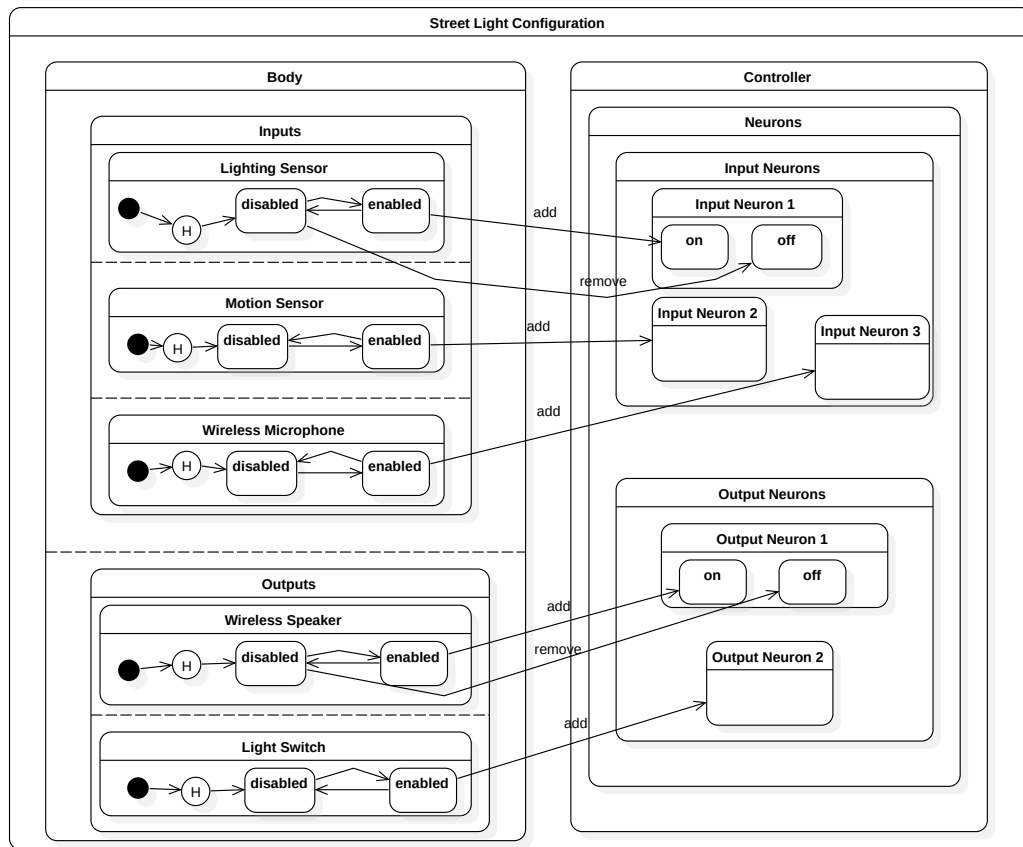


Figure 4.6: The designed autonomous street lights exploiting the proposed reference model for the Body and Controller Configuration components.

This section introduces an illustrative example to show an application

of the proposed reference model. In particular, we illustrate the statechart of embodied agents in a specific application scenario: autonomous street lights, in which each street light represents an embodied agent. In this scenario, each street light may contain a lighting sensor, motion sensor, wireless microphone, wireless speaker and light switch apparatus. As we describe in Figure 4.6, configuring the agent body consists of disabling or enabling some inputs and outputs. For example, we may have a street light agent containing only a lighting sensor as its input, and a light switch apparatus as its output; or we can create more robust street light agents by enabling them to communicate among themselves by means of wireless gadgets.

Configuring the agent controller also consists of disabling or enabling some components. As shown in Figure 4.6, the number of neurons varies according to the sensors and actuators that were enabled. Accordingly, there will be an input neuron for each activated sensor and an output neuron for each activated output. To simplify this figure, we did not illustrate the Connections component shown in Figure 4.3. Basically, if a neuron is at the "on" state (i.e. it is active), it will be able to be connected to other neurons. For example, an output neuron can connect to an input neuron, consequently configuring a recurrent neural network.

As we previously described, agent's behavior is a consequence of the body and controller configuration, and the environment perturbation. If a specific sensor is enabled, the street light will be able to sense its specific variable, as we described in Figure 4.4. For example, as shown in Figure 4.7, if the motion sensor is enabled, the agent will be able to calculate the flow of people in the environment. In the same way, if it has a wireless microphone, it will be able to receive communication signals from the other street lights.

Based on its outputs, the wireless speaker and the light switch apparatus, actions that can be taken by this agent are: "conversing with neighboring streetlights" and "activating light". For example, if an agent is able to activate the light, how many levels of brightness can this agent generate? As shown, to activate the light, the agent must select one of the following substates: OFF, DIM or ON. This selection depends on the results originated at the Decision state, as we explained in subsection 4.3.2.

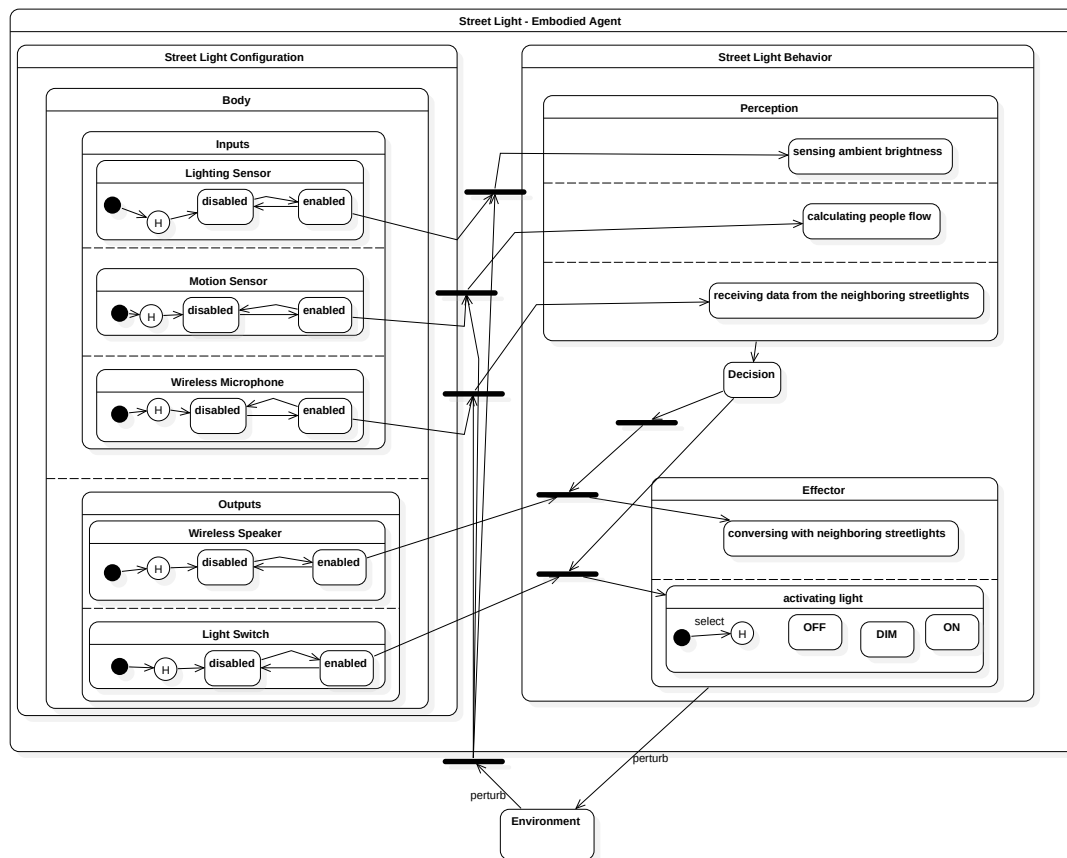


Figure 4.7: The designed autonomous street lights exploiting the proposed reference model for the Agent's Behavior component.

5

Approaches and their Applications

In this chapter, we present the approaches that we created to design and test reconfigurable embodied agents based on our proposed reference model. In accordance with the questions that we raised in Section 1.3, this chapter has the following sections: one to describe the (i) software framework for the development of embodied agents to IoT applications; one to describe the (ii) architecture to configure embodied agents according to the environment in which they are situated; and another one for the (iii) approach for testing embodied agents. Each one of the sections has a description of the approach and an evaluation subsection.

In each section, we summarize how the proposed approach adheres to the proposed reference model by refining the main statechart components.

5.1

Framework for IoT Embodied Agents

In this section, we answer the question **RQ2. How to design and implement a software framework to support the development of embodied agents?**. Part of the content of this section is published in (Nascimento and Lucena, 2017).

Based on the Google Trends tool (Google, 2018), the Internet of Things (IoT) (Atzori et al., 2012) is emerging as a topic that is highly related to physical agents and machine learning. In fact, the use of learning agents has been proposed as an appropriate approach to modeling IoT applications (Nascimento and Lucena, 2017). These types of application address the problems of distributed control of devices that must work together to accomplish tasks (Atzori et al., 2012). This has caused agent-based IoT applications to be considered for several domains, such as health care, smart cities, and agriculture. For example, in a smart city, software agents can autonomously operate traffic lights (Nascimento and Lucena, 2017; Santos et al., 2017), driverless vehicles (Herrero-Perez and Martinez-Barbera, 2008) and street lights (Nascimento and Lucena, 2017).

We developed a novel software framework to instantiate different appli-

cations based on IoT embodied agents, named the Framework for the Internet of Things (FIoT) (Nascimento and Lucena, 2017). Using this framework, we prototyped four IoT agent-based applications (Nascimento et al., 2015; Nascimento Marx Leles Viana, 2016; Nascimento and Lucena, 2017; Nascimento and Lucena, 2017).

5.1.1

IoT Embodied Agents

According to the description about embodied agents provided in Section 2.2, Figure 5.1 illustrates an IoT embodied agent in a scenario of autonomous cars. In this example, the body of the agent is a car with four wheels, GPS, headlights, etc. As described above, an embodied agent must have a local analysis architecture to sense the environment and behave accordingly. In such example, the autonomous car uses an artificial neural network. There is an input neuron for each one of the car's sensors and an output neuron for each one of the motors and actuators. The neuron output values may determine the direction of the wheels and whether the car turns on the headlights.

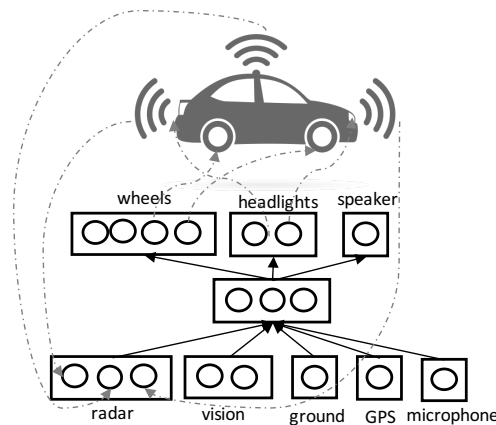


Figure 5.1: An example of an IoT embodied agent.

5.1.2

Description of the Framework

We developed an agent-based model as a foundation for developing different kinds of application for IoT. Our approach is completely based on MAS and artificial intelligence paradigms such as neural networks and evolutionary algorithms. Our goal is to provide mechanisms that recognize and manage things in the environment automatically. As depicted in Figure 5.2, our model consists of three layers: physical (L1), communication (L2), and application (L3). Each thing in the environment (physical layer) is recognized and controlled by agents in the application layer.

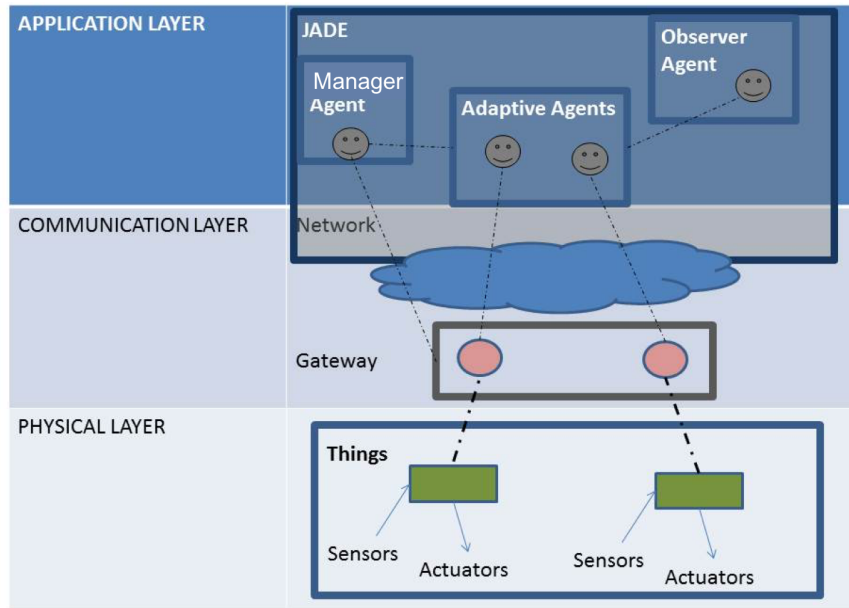


Figure 5.2: An agent-based model to generate IoT applications.

The physical layer consists of simulated or real devices (also named smart things/objects) and environments. In order to model the physical layer, the project designer has to define the features of smart things as well as the features of their surrounding environment. The designer must decide on the environmental conditions (i.e. the variables of the Environment component of the embodied agent statechart) that need to be monitored such as temperature, relative humidity or traffic. Once these conditions are chosen the designer can specify performance criteria for the smart things that collect data or make changes to the environment.

The communication layer specifies the communication among agents in the application layer. Each smart thing has one address, so an agent can access this address to obtain and set the necessary information to control the thing. We suggest the Java Agent Development Framework (JADE) (Bellifemine et al., 2007) and its variants (JADEX, LEAP) to implement the communication layer among agents and smart things in order to address heterogeneous devices such as PCs, PDAs, resource constrained-devices or Smartphones.

Agent-Based Model

The application layer uses a MAS to provide services, such as collecting, analyzing and transmitting data from several sensors to the Internet and back again. Basically, FIoT supports the development of three types of agents: (i) Manager Agents; (ii) Adaptive Agents; and (iii) Observer Agents. The

primary role of the Manager Agent is to detect new things that are trying to connect to the system and make that connection. Adaptive Agents control things at the scenario and must execute three key activities in sequence, namely: (i) collect data from the thing (i.e. perception); (ii) make decisions; and (iii) take actions (i.e. effector). In short, an Adaptive Agent represents the behavior of an embodied agent, executing the actions described in the Behavior component of the embodied agent statechart (see statechart 4.4). The Observer Agent examines the environment to determine if the system is meeting its global goals, executing the actions of the “Task Evaluation” component (see statechart 4.5). When the actions of agents are far from what an Observer Agent expects, it executes a supervised or unsupervised learning method, such as back-propagation or a genetic algorithm, to adjust agents’ controller.

Agent Reconfiguration

The process of adjusting an agent controller consists of generating a new set of weights for the neural network that has been used by the embodied agents, and then evaluating how agents will behave in the environment. The Observer Agent sets their neural networks with the configuration that conforms to the embodied agents’ desired global action. While the Observer Agent evaluates the group of agents and optimizes this neural controller accordingly, we have a group of embodied agents using the neural controller to take decisions.

In this current version, this agent could change only the neural weights. Based on the proposed statecharts, our further goal is to extend the Observer-Agent in order to allow it to change at runtime the whole ANN’s structure. Therefore, during an experiment, a collection of embodied agents can start the experiment, for example, using a neural network with three sensors, two hidden layers and a hyperbolic tangent function as the transfer function of all neurons; and finish this same experiment using a totally different neural network, with one sensor, one hidden layer and a sigmoid function.

The Observer Agent is tightly coupled to the application being developed. The evaluation process has to be implemented according to the expected global solution. For example, if an application for automobile traffic control has the goal of reducing urban traffic congestion, the evaluation may be performed based on the number of vehicles that had finished their routes in a specific period. Another variable activity is the generation of new configurations for neural networks, which depends on the applied adaptation technique.

Details of FIoT

As presented in subsection 5.1.2, our model proposes the use of JADE to support the communication among software agents and devices at the physical layer. FIoT extends JADE, a Java framework to implement MAS through the development of JADE agents, the behavior of agents, the controller to be used by Adaptive Agents, and the adaptive process to be executed by the Observer Agent. In addition, the system gives support to different interface communication message systems, such as sockets and ACL. We present the key FIoT classes (Sommerville, 2004) of the main packages.

The class diagram depicted in Figure 5.3 illustrates the FIoT classes associated with the creation of agents and their execution loops. As described before, the FIoT agent classes are the ManagerAgent, ObserverAgent and AdaptiveAgent classes, which extend the FIoTAgent class. Then, FIoT agents can access and make changes to the list of controllers (ControllerList class). This list stores all controllers already created by the ManagerAgent for each type of smart thing, such as a chair with one temperature sensor, lamp with one presence sensor and one alarm actuator.

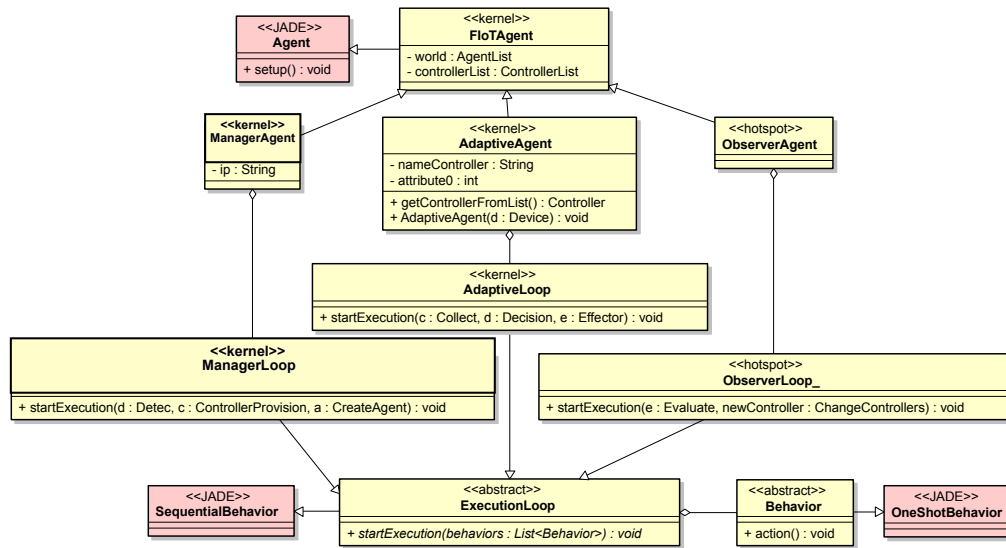


Figure 5.3: Class diagram of FIoT - Agents.

All agents execute sequential behaviors, named as ExecutionLoop: ManagerLoop, AdaptiveLoop and ObserverLoop classes. The sequential behavior is a JADE behavior that supports the composition of activities (Bellifemine et al., 2007). Thus, the ExecutionLoop is a sequence of smaller actions. For example, for Adaptive Agents, these execution loops are composed of collect, decide and effect activities.

The class diagram depicted in Figure 5.4 illustrates the collection of behaviors already developed. Activities such as evaluation and controller adaptation are examples of hot spots. Thus, new strategies for evaluation and adaptation can be developed to be used by agents. The Manager Agent’s execution loop performs three behaviors: “Detect,” “CreateAgent,” and “ControllerProvision.”

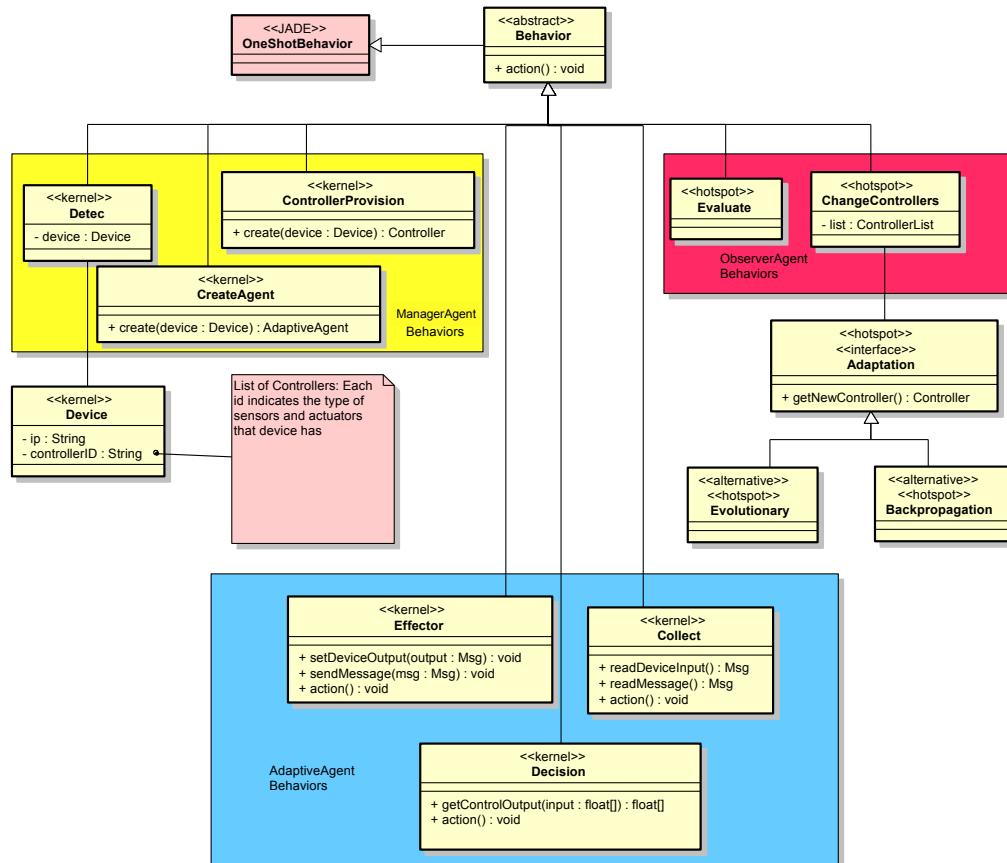


Figure 5.4: Class diagram of FIoT - Behaviors.

While ObserverAgents access the ControllerList to adapt controller configurations through ChangeControllers behavior, an AdaptiveAgent uses the ControllerList to get its controller, set data input, and obtain the calculated output.

The class diagram depicted in Figure 5.5 illustrates the controller classes. Agents as virtual homogeneous things can use the same controller to make decisions. For example, where similar smart lamps have to be managed, the same ANN controller can be used by Adaptive Agents. The ManagerAgent stores the smart lamp controller in ControllerList as “lampNeuralNetwork.” If there is another group of devices, the ManagerAgent has to use a different controller.

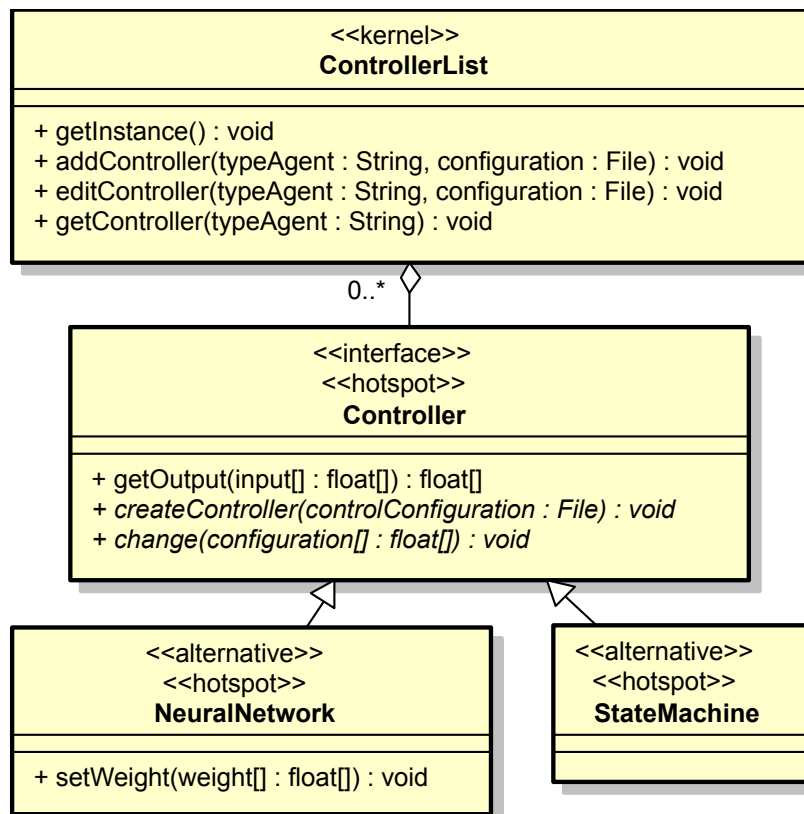


Figure 5.5: Class diagram of FIoT - Controllers.

5.1.3

Instances of the FIoT

The frozen spots are part of the FIoT kernel and each of the proposed applications will have the following modules in common:

- Detection of devices by the ManagerAgent
- The assignment of a neural network to a particular Adaptive Agent by the ManagerAgent
- Creation of Agents
- Data Collection execution by Adaptive Agents
- Making decisions by Adaptive Agents
- Execution of effective activity by Adaptive Agents
- The communication structure among agents and devices

Some features are variable and may be selected/developed according to the application type, as follows:

- Agent controller configuration
- Evaluation by the Observer Agent

- Neural Network adaptation by the Observer Agent

Thus, to create an FIoT instance, a developer has to implement/choose: (i) a neural network; (2) an adaptive technique to train the neural network; and (iii) an evaluation process such as a genetic algorithm that performs evaluation via a fitness function. We implemented FIoT to support the use of different neural network types, since we provided an abstract controller class. For example, a framework user can implement a recurrent neural network (a type of neural network that propagates data forward, but also backwards) and use an evolutionary algorithm to evolve its structure and transition probabilities. Thus, it is possible to generate applications using different configurations. A framework user should select a configuration that works better toward solving a given problem.

5.1.4

Evaluation: Examples of Applications

We evaluate FIoT by implementing its hot spots or flexible points to generate three different applications. We consider the following IoT applications or instances in the FIoT evaluation process: (i) quantified fruits; and (ii) smart street light. This section presents a brief description of each example by completing the hot spots and illustrating how the generated applications adhere to the proposed framework.

5.1.5

Application I: Quantified Fruits

As suggested in (Swan, 2012), the concept of *Quantified Self* (QS) represents the capacity for connected objects to self-measure and self-monitor their human owner. Examples are connected watches or phones that measure heart rate, pressure, exercising habits, etc. Capacities for analysis, patterns detection and prediction (using statistical analysis and machine learning techniques) may be included in order to infer personalized monitoring and diagnostic, e.g., for health monitoring.

In (Briot et al., 2016), we investigate the adaptation of this idea to arbitrary things, therefore named *Quantified Things* (Nascimento et al., 2015). Some early examples are Quantified Cars (Swan, 2015), using the large electronic monitoring and control facilities of a car to monitor, diagnose and control various features of a car. Swan suggests the use of “QS car chips” to collect cars automotive data and store these informations in a cloud database. By using a mobile application, users could have access to

their car's information, such as maintenance records, suggested and scheduled maintenance, and take more accurate action as a result.

We have decided to address the case of agriculture food products. In particular, the lifecycle of fruits has an important impact on its economy. An important issue is indeed to minimize the loss of fruits too mature to be consumed and at the same time to minimize the risk of shortage of products for the consumers. This is specially true in the case of bananas, a fruit having a relatively short ripening period, and very much depending on various conditions (temperature, humidity, light, aeration) (Johnson et al., 2008). Therefore, important decisions must be taken at various steps of the lifecycle: when to best harvest the fruits, depending of the expected travel (type and duration) to the consumer, how to best transport them, how to store them, at a large scale in a storage or a grocery store, down to the consumer house, etc.

In order to explore these issues, we have designed a prototype multi-agent architecture for Quantified Fruits. Its objective is self-monitoring and self-prediction of fruit maturation. We have tested the architecture in the case of bananas and have evaluated it as a proof of concept. A design for this architecture is presented using FIoT, as depicted in Figure 5.6.

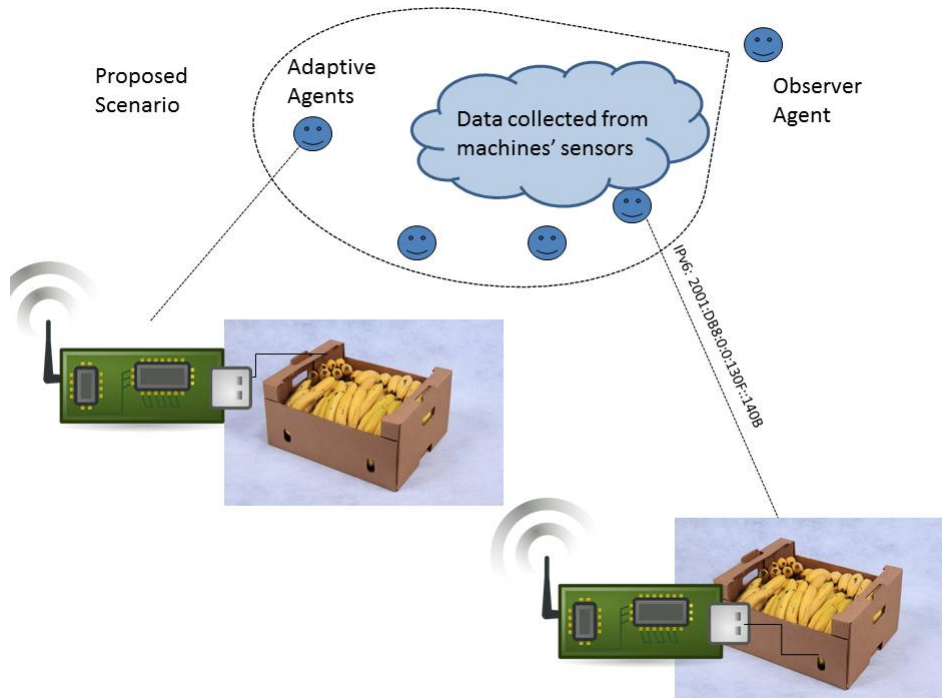


Figure 5.6: An instance of FIoT to create “Quantified Fruit.”

Table 5.1 shows how the “Quantified Fruit” application adheres to the proposed framework by extending the FIoT flexible points. The hot spot “making evaluation” is developed for this application as an individual

evaluation. The Observer Agent maintains a data set containing input from Adaptive Agents and neural predictions. Based on this historical data, for each adaptive agent execution, the Observer Agent evaluates if an individual result requires a collective adaptation.

Table 5.1: Case I: Flexible Points

Framework	Application
Controller	Three Layer Neural Network
Making Evaluation	Individual Evaluation: for each agent evaluation, the Observer Agent concludes if all Adaptive Agents need to adapt or not
Controller Adaptation	Supervised Learning (Backpropagation)

Agent Body

Embodied agents in this application encapsulate an Arduino microcontroller (Arduino) and its 5 sensors, respectively measuring: methane, hydrogen, temperature, humidity, and light. We have selected these sensors based on some works (Boe and Salunkhe, 1967; Johnson et al., 2008) that investigate various factors that interfere on fruit's perishability.

Agent Controller

The agent controller encapsulates an artificial neural network (ANN) used for prediction. We have decided to use an artificial neural network (ANN) architecture for the prediction module. The reason is as follows: ANNs are well known architectures and they have proven their efficiency and moreover versatility. As opposed to linear or polynomial regression modules where one has to *a priori* select a model (linear, quadratic, cubic, including product of features, etc.), the model of a neural network is generic enough although some configuration has to be decided (e.g. the number of hidden layers, the number of units of the hidden layer(s)).

The neural network includes an input layer with 5 units (corresponding to the 5 parameters produced by the 5 sensors), one hidden layer with 4 units and an output layer with one unit (corresponding to the number of days predicted).

Task Evaluation - Observer Agent

The Observer Agent encapsulates both backpropagation and prediction error minimization algorithms for training the neural network (Rumelhart et al., 1986). In addition, it also encapsulates a data base containing the various data of the experiments, that is shared by all agents.

Our current method for training the neural network (adjusting the weights of the neuron connexions in order to minimize the error (differences) between predicted and target values) is quite standard: 1) using backpropagation algorithm (to compute the gradients) (Rumelhart et al., 1986); 2) combined with an algorithm to minimize the cost function (prediction error) – we have tried out batch gradient descent as well as generic optimization algorithms (from off-the-shelf libraries).

Note that, in addition to traditional off-line learning approach, we also experimented with a (simplified) incremental learning approach, where **Adaptive Agent** pro-actively self-assesses its prediction accuracy and if necessary requests **ObserverAgent** to incrementally update its prediction model by launching a new learning phase on the new example(s) (in a similar way to on-line learning).

Experimental Setting

The user will try various ways for storing a banana, taking four condition parameters into account: (i) dark (i.e. in a closed or open box); (ii) room (i.e. the box being stored in a fridge or at room temperature); (iii) rotten fruit (i.e. in a box alone or putting together with a rotten); and (iv) ripe fruit (i.e. putting together with a ripe fruit).

Below, we detail four of the possible settings for storing a banana (summarized at Table 5.2 and depicted at Figure 5.7):

(a) In an open box, at room temperature, alone; (b) In an open box, together with a rotten fruit; (c) In the fridge, with a ripe fruit; (d) In a closed box, at room temperature, with a rotten fruit.

Table 5.2: Configuration of experiments at Figure 5.7.

Experiment	Box		Room		Rotten Fruit		Ripe Fruit	
	<i>open</i>	<i>close</i>	<i>room</i>	<i>fridge</i>	<i>yes</i>	<i>no</i>	<i>yes</i>	<i>no</i>
(a)	X		X			X		X
(b)	X		X		X			X
(c)	X			X		X	X	
(d)		X	X		X			X

For each setting, a user creates a new experiment on his smartphone user interface. Then, he triggers the measurement of the parameters (light,

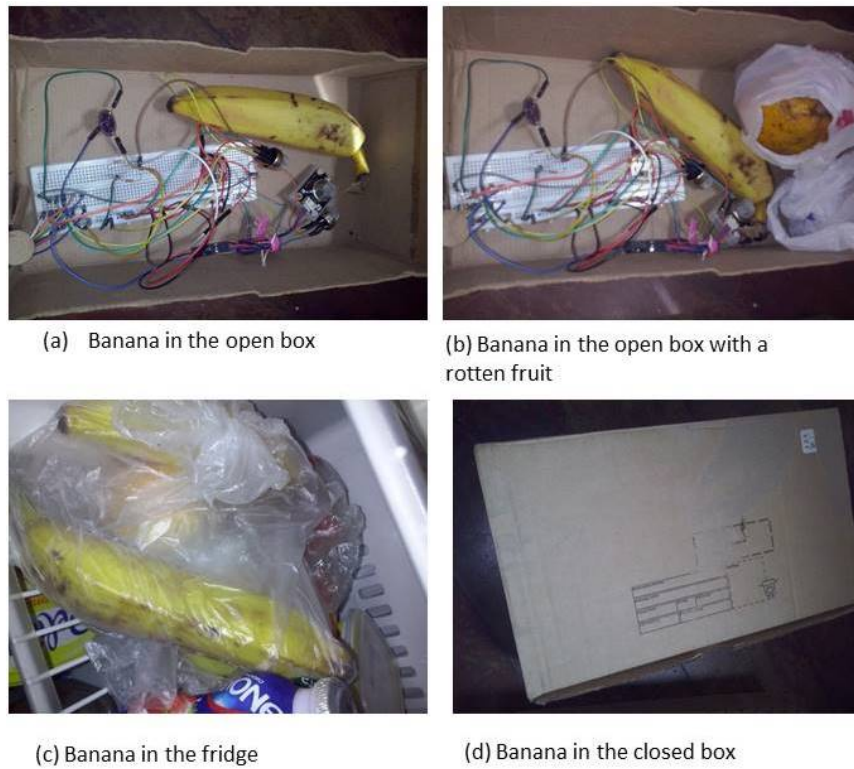


Figure 5.7: Examples of scenarios.

temperature, methane, hydrogen and humidity), which will be recorded in the database. He then later checks (usually every day) the maturation of the fruit and reports on the interface when the starting maturation occurs. This process was essential to elaborate an initial database to improve system's predictions. In practice, we have conducted several experiments in parallel, putting a dozen of bananas in different settings and monitoring in parallel their respective maturation process.

Following standard methodology in machine learning, we have partitioned our dataset into a training set and a testing set.

Training Set

Table 5.3 shows a subset of the training set used, which represents the data collected from the experiments illustrated at Figure 5.7. At the beginning of each experiment, the embodied agent collects the measured values: temperature (abbreviated Temp.), which is registered in Celsius (C), relative humidity (RH), hydrogen gas (Hyd.), methane gas (Met.), and luminosity (Lum.). Values of gas sensors are recorded according to the sensor output value (V.). At the end of each experiment, the user reports the “actual” fruit lifespan (this information is subjective since in current experiments naked-eye observation determines it).

Table 5.3: Subset of the training set.

Temp	RH	Hyd	Met	Lum	Lifespan
27.62	70.22	2	184.0	15.0	14
28.02	72.53	8	275.0	10.0	5
27.81	72.75	3.0	258.0	3.0	10

Test Set

Table 5.4 shows results for a subset of the test set. This example, which was performed outside the fridge and in an open box, shows a good prediction. The system predicted thirteen days, and the user reported that the banana spoiled in approximately twelve days.

Table 5.4: Subset of the test set.

Temp	RH	Hyd	Met	Lum	Lifespan	
					<i>Observed</i>	<i>Predicted</i>
28.21	70.24	3.0	183.0	16.0	12	13

Experimental Results: Measuring Accuracy

We believe that these first experiments are promising, the prediction module showing good prediction accuracy. Obviously, we need to conduct more experiments with different settings to collect more data.

We have conducted some analysis of our prediction module. Figure 5.8 shows the validation curve, which compares the evolutions of the prediction error for the training set (we will name it *training error*, depicted in a blue solid line) and of the prediction error for the cross validation set (*cross validation error*, depicted in a green dashed line) for various (increasing) values of λ (the regularization parameter used to control overfitness). The figure shows that 0.01 is a good value for λ as cross validation error is minimal. For a smaller value, there is some variance (overfitness) because the training error is almost null and the cross validation error is significant, showing the poor generalization of the model. For a larger value, the cross validation error is increasing (note that the training error is also increasing), showing an increasing bias (underfitness).

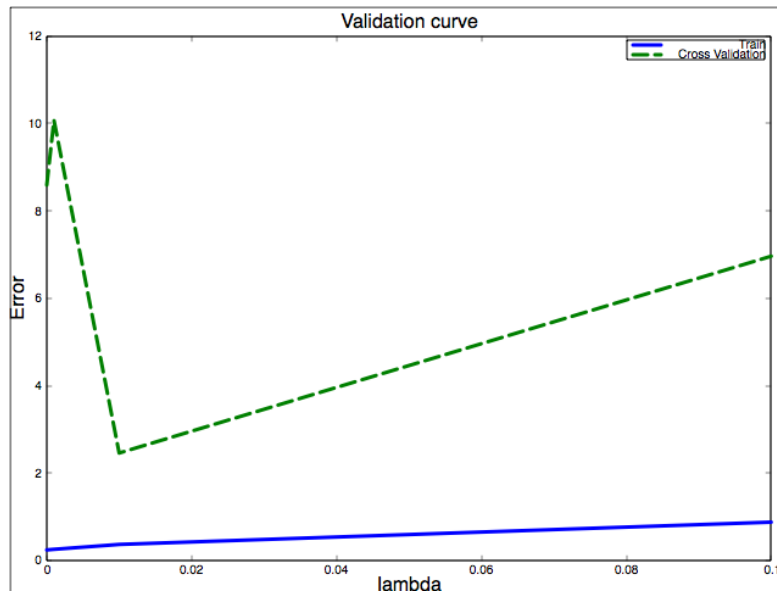


Figure 5.8: Validation curve.

Figure 5.9 shows the learning curve, i.e., the evolution of prediction error depending on the size of the training set. The figure shows that the training error is almost null and that the cross validation error stays low, confirming that the model has low bias and low variance. These preliminary analyses are encouraging. We are conducting more experiments in order to collect more data in order to further improve the model.

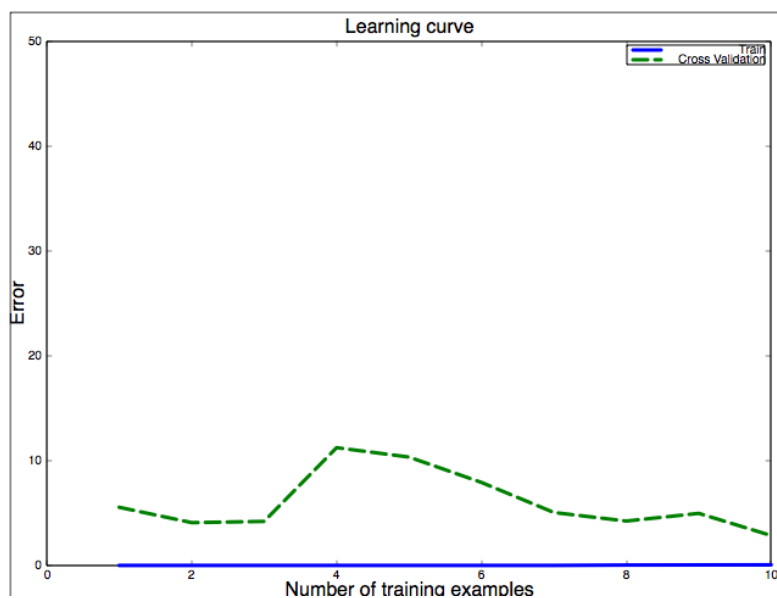


Figure 5.9: Learning curve.

5.1.6

Application II: Smart Street Lights

In order to evaluate our proposed framework to create Internet of Things applications based on embodied agents, we developed a smart street light application.

The overall goal of this application is to reduce the energy consumption and maintain the maximum visual comfort in illuminated areas. For this purpose, we provided each street light with ambient brightness and motion sensors, and an actuator to control its light intensity. In addition, we also provided street lights with wireless communicators. Therefore, they are able to cooperate with each other in order to establish the most evaluable routes of the passers-by and to achieve the goal of minimizing energy consumption.

We used an evolutionary algorithm to support the design of this system's features automatically. By using a genetic algorithm, we expect that a policy for controlling the street lights, with a simple communication system among them, will emerge from this experiment. Therefore, no system feature such as the effect of ambient brightness on light status changes was specified at design-time.

The training process can occur in a simulated or in a physical environment. However, many devices could be damaged if we were to use real equipment, since several configurations must be tested during the training process. Therefore, to execute the training algorithm, we decided to simulate how smart street lights behave in a fictitious neighborhood. After the training process, we transferred the evolved neural network to physical devices and observed how they behaved in a real scenario.

Table 5.5 summarizes how the “Street Light Control” application will adhere to the proposed framework, while extending the FIoT flexible points.

Table 5.5: FIoT's Flexible Points

FIoT Framework	Street Light Control Application
Controller	Three Layer Neural Network
Making Evaluation	Collective Fitness Evaluation: Test a pool of candidates to represent the network parameters. For each candidate, it evaluates the collection of smart street lights, comparing fitness among candidates
Controller Adaptation	Evolutionary Algorithm: Generate a pool of candidates to represent the network parameters

Experimental Setting

In this subsection, we describe a simulated neighborhood scenario. Figure 5.10 depicts the elements that are part of the application namely, street lights, people, nodes and edges. We modeled our scenario as a graph, in which a node represents a street light position and an edge represents the smallest distance between two street lights.



Figure 5.10: Simulated Neighborhood.

The graph representing the street light network consists of 18 nodes and 34 edges. Each node represents a street light. In the graph, the yellow, gray, black and red triangles represent the street light status (ON/DIM/OFF/Broken Lamp). Each edge is two-way and links two nodes. In addition, each edge has a light intensity parameter that is the sum of the environmental light and the brightness from the street lights in its nodes. Our goal is to simulate different lighting in different neighborhood areas.

People walk along different paths starting at random departure points. Their role is to complete their routes, reaching a destination point. A person can only move if his current and next positions are not completely dark. In addition, we also supposed that people walk slowly if the place is partially devoid of light. For simulation purposes, we chose four nodes as departure points (yellow nodes) and two as destinations (red nodes). We started with

ten people in this experiment. We also configured that 20% of the street lights lamps will go dark during the simulation.

Agent Body

Each street light in the simulation has a micro-controller that is used to detect the approximation of a person, interact with the closest street light, and control its lights. A street light can change the status of its light to ON, OFF or DIM, as depicted in Figure 5.11.

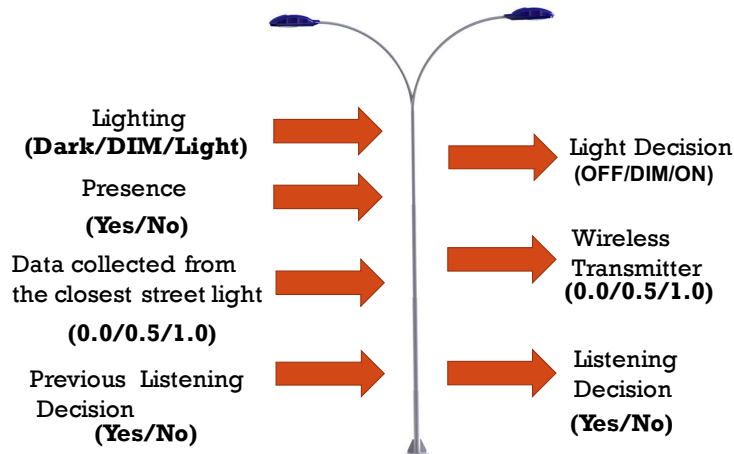


Figure 5.11: Variables collected and set by streetlights.

Smart street lights have to execute three sequential tasks: data collection, decision-making and action enforcement, as depicted in Figure 5.12. The first task consists of receiving data related to people flow, ambient brightness, data from the neighboring street lights and current light status. The second task consists of analyzing collected data and making decisions about actions to be enforced. To make decisions, smart street lights use a three-layer feedforward neural network with a feedback loop (Haykin, 1994). Feedback occurs because one or more of the neural network's outputs influence the next neural network's inputs. The last task is the action enforcement, which consists of setting the value of output variables, such as: wirelessTransmitter, a signal value to be transmitted to neighboring agents; and lightDecision, that activates the light's OFF/DIM/ON functions.

Agent Controller

We instantiated FIoT's controller by implementing a three-layer neural network controller for our smart street lights (see Figure 5.13).

The input layer includes four units that encode the activation level of sensors and the previous output value of listeningDecision. The output layer contains three output units: (i) listeningDecision, that enables the smart

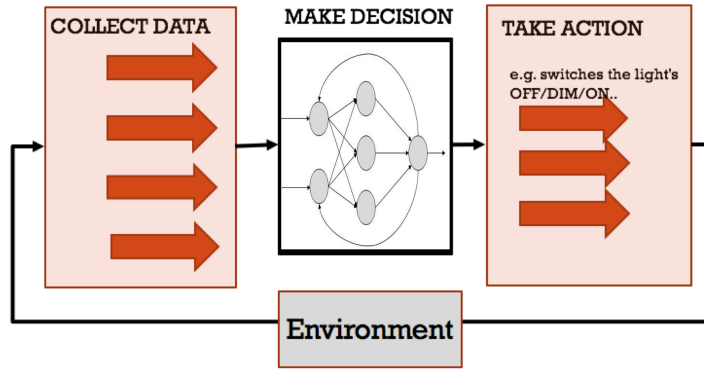


Figure 5.12: A streetlight uses a neural network to make decisions based on collected data.

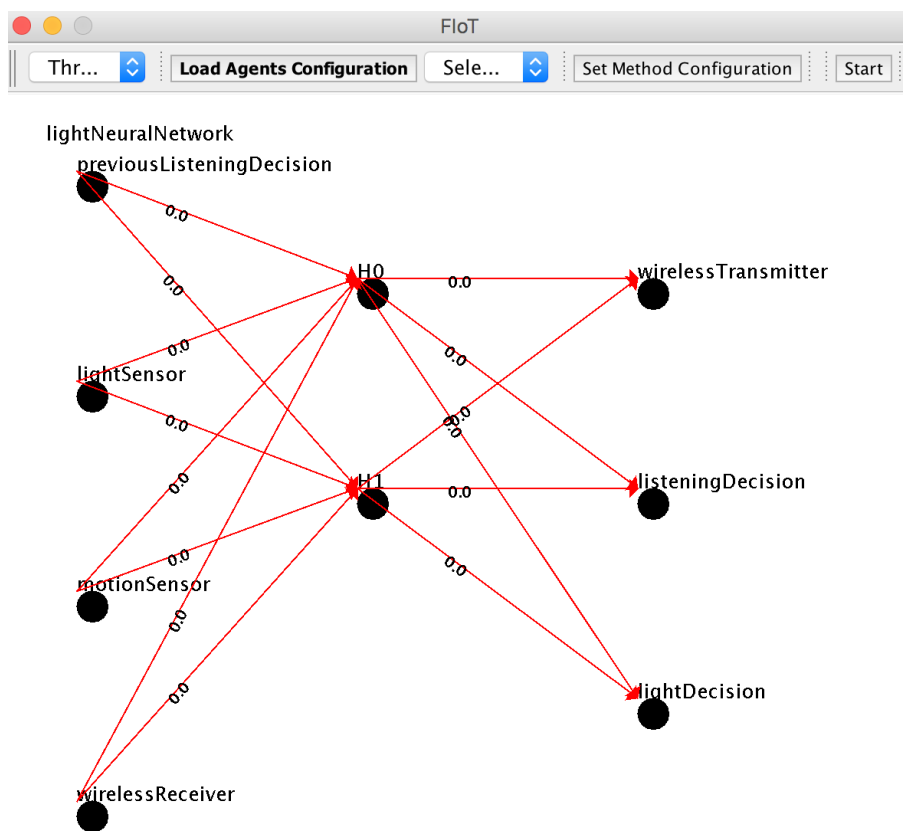


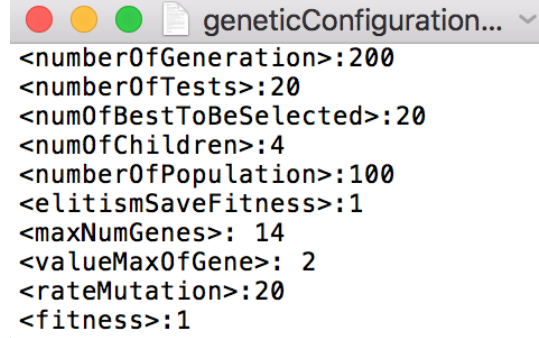
Figure 5.13: The neural network controller for smart street lights: zeroed weights (FloT's Application View).

lamp to receive signals from neighboring street lights in the next cycle; (ii) `wirelessTransmitter`, a signal value to be transmitted to neighboring street lights; and (iii) `lightDecision`, that switches the light's OFF/DIM/ON functions.

The middle layer of the neural network has two neurons connecting the input and output layers. These neurons provide an association between sensors and actuators, which represent the system policies that can change based on the neural network configuration.

Task Evaluation and Training the Neural Network

The weights in the neural network used by the smart street lamps vary during the training process, as the system applies a genetic algorithm to find a better solution. Figure 5.14 depicts the simulation parameters that were used by the evolutionary algorithm. We selected these parameters values (i.e number of generation and tests, population size, mutation rate, etc.) according to known experiments of evolutionary neural networks that we found in the literature (Marocco and Nolfi, 2007; VIDE and Nolfi, 2006).



```

<numberOfGeneration>:200
<numberOfTests>:20
<numOfBestToBeSelected>:20
<numOfChildren>:4
<numberOfPopulation>:100
<elitismSaveFitness>:1
<maxNumGenes>: 14
<valueMaxOfGene>: 2
<rateMutation>:20
<fitness>:1

```

Figure 5.14: Configuration file to evolve the neural network via genetic algorithm using FIoT.

During the training process, the algorithm evaluates the weight possibilities based on the energy consumption, the number of people that finished their routes after the simulation ends, and the total time spent by people to move during their trip. Therefore, each weights set trial is evaluated after the simulation ends based on the following equations:

$$pPeople = \frac{(completedPeople \times 100)}{totalPeople} \quad (5-1)$$

$$pEnergy = \frac{(totalEnergy \times 100)}{\left(\frac{11 \times (timeSimulation \times totalSmartLights)}{10}\right)} \quad (5-2)$$

$$pTrip = \frac{(totalTimeTrip \times 100)}{\left(\left(\frac{3 \times timeSimulation}{(2)}\right) \times totalPeople\right)} \quad (5-3)$$

$$fitness = (1.0 \times pPeople) - (0.6 \times pTrip) - (0.4 \times pEnergy) \quad (5-4)$$

in which $pPeople$ is the percentage of the number of people that completed their routes as of the end of the simulation out of the total number of people in the simulation; $pEnergy$ is the percentage of energy that was consumed by street lights out of the maximum energy value that could be consumed during the simulation. We also considered the use of the wireless transmitter to calculate energy consumption; $pTrip$ is the percentage of the total duration time of people's trips out of the maximum time value that their

trip could spend; and *fitness* is the fitness of each representation candidate that encodes the neural network.

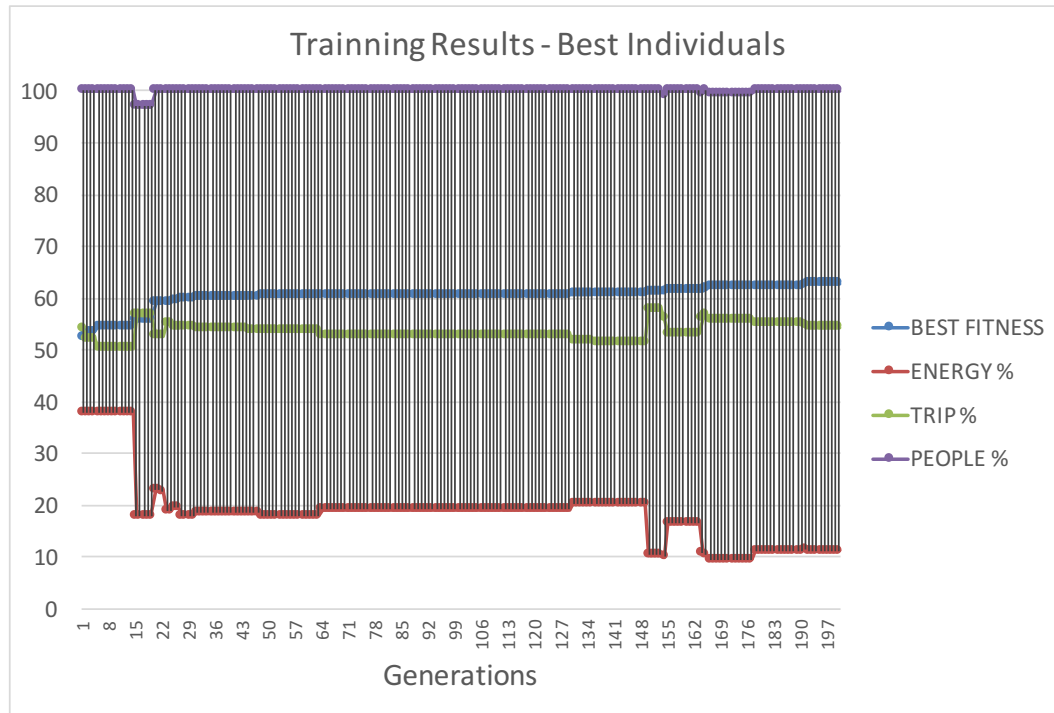


Figure 5.15: Simulation results - Most-Fit from each generation.

Normally, the performance of the most-fit individual is better than the others. Figure 5.15 illustrates the best individual from each generation (i.e. the candidate with the highest fitness value). As shown, the best individuals from the generations tend to minimize energy consumption and find an equilibrium between energy consumption and the trip time. We selected the best individual from the last generation to investigate its solution, as shown in the subsection (5.1.6).

Experimental Results: Evaluating the Best Candidate

After the end of the evolutionary process, the algorithm selects the set of weights with the highest fitness (equation 5-4). Figure 5.16 depicts the evolved neural network configured with the best set of weights found during the evolution.

One disadvantage of using neural networks combined with evolutionary algorithms is to understand and explain the behaviors that were automatically assigned by the smart things. Therefore, we executed the simulated street lights using the evolved network in order to generate logs and extract the rules that are implicit in patterns of the generated input-output mapping. To generate these logs, we used the runtime monitoring platform that we proposed

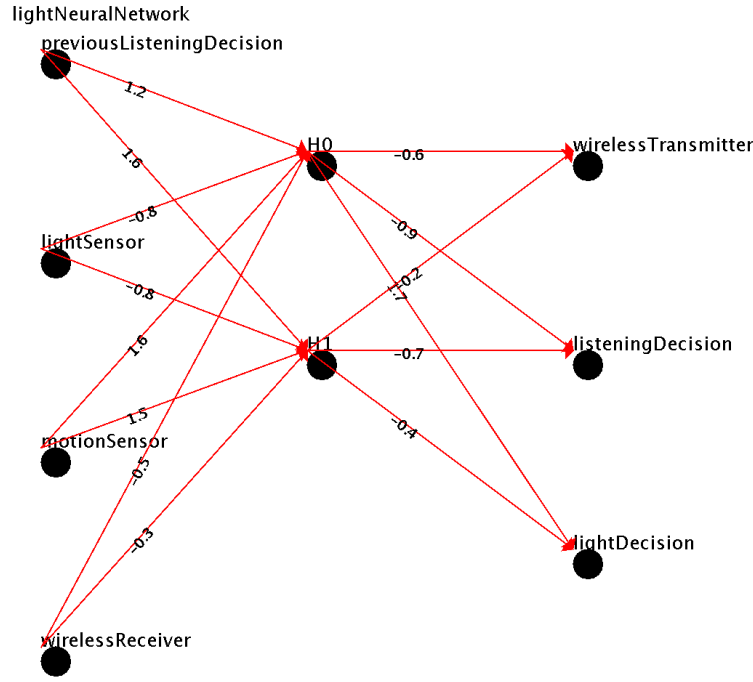


Figure 5.16: The Evolved Neural Network to be used as a controller for real Street Lights (FIoT's Application View).

in (Nascimento et al., 2017) to test distributed systems, as described in Section 5.3.

After analyzing logs, we could realize the rules that were created by the evolved neural network in order to understand why street lights decided to communicate and switch the lights ON. The code below exemplifies some of these rules:

$$(I_0 = 1.0 \wedge I_1 = 0.5 \wedge I_2 = 0.0 \wedge I_3 = 0.0) \Rightarrow (Out_0 = 0.0 \wedge Out_1 = 1.0 \wedge Out_2 = 0.0) \quad (5-5)$$

$$(I_0 = 1.0 \wedge I_1 = 0.5 \wedge I_2 = 1.0 \wedge I_3 = 0.0) \Rightarrow (Out_0 = 0.0 \wedge Out_1 = 1.0 \wedge Out_2 = 0.5) \quad (5-6)$$

$$(I_0 = 0.0 \wedge I_1 = 0.0 \wedge I_2 = 0.0 \wedge I_3 = 0.0) \Rightarrow (Out_0 = 0.5 \wedge Out_1 = 0.0 \wedge Out_2 = 0.0) \quad (5-7)$$

$$(I_0 = 1.0 \wedge I_1 = 0.0 \wedge I_2 = 0.0 \wedge I_3 = 0.5) \Rightarrow (Out_0 = 0.0 \wedge Out_1 = 1.0 \wedge Out_2 = 0.5) \quad (5-8)$$

in which the variables are:

$$\begin{aligned}
 I_0 &\equiv \text{previousListeningDecision}, I_1 \equiv \text{lightSensor}, \\
 I_2 &\equiv \text{motionSensor}, I_3 \equiv \text{wirelessReceiver}, \\
 O_0 &\equiv \text{wirelessTransmitter}, O_1 \equiv \text{listeningDecision}, \\
 O_2 &\equiv \text{lightDecision}
 \end{aligned}
 \tag{5-9}$$

Based on the generated rules and the system execution, we could observe that only the street lights with broken lamps emit “0.5” by its wireless transmitter (rule 5-7). In addition, we also observed that a street light that is not broken switches its lamp ON if it detects a person’s approximation (rule 5-6) or receives “0.5” from wireless receiver (rule 5-8) .

Prototyping the Smart Street Light Device

As depicted in Figure 5.17, the prototype of the smart street light is composed of an Arduino (Arduino) and the following sensors and actuators: (i) HC-SR501 (a device that detects moving objects, particularly people. The detection distance is slightly shorter - maximum of 7 meters); LM393 light sensor (a device to detect the ambient brightness and light intensity); nRF24L01 (a wireless module to allow one device to communicate with another); and (iii) LEDs (the representation of a lamp).

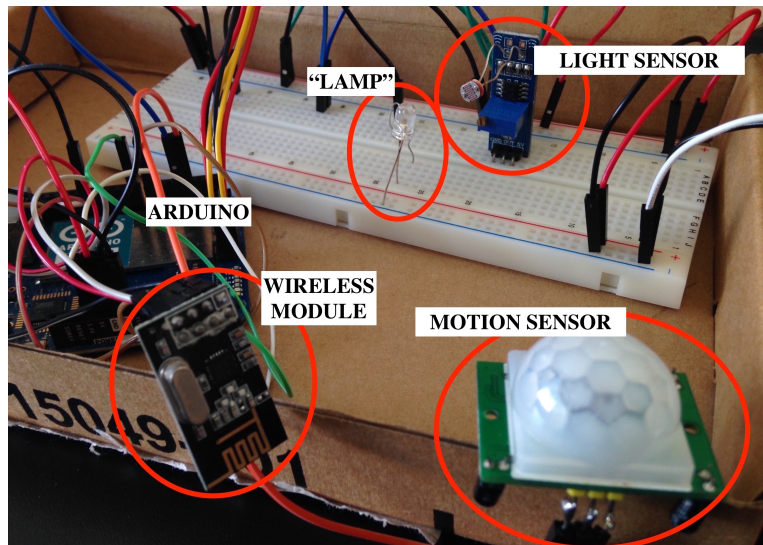


Figure 5.17: Prototyping the smart street light.

We put two LEDs in this circuit. Our goal is to simulate light intensity. Therefore, if a smart street light decides to set its light intensity to the maximum, both LEDs will be on. If the light intensity is medium, one LED will be on and the other LED will be off.

Transferring the evolved neural network to physical devices

After the neural network has been evolved, we codified it into the Arduino. We show below the code in C++ language that operates as a neural network inside the Arduino:

```
double fSigmoide(double x){
double output = 1 / (1 + exp(-x));
return output;
}
```

```
double calculateHiddenUnitOutput(double w[4]){
double H = previousListeningDecision*w[0] +
lightSensor*w[1]+motionSensor*w[2]+wirelessReceiver*w[3];
double HOutput = fSigmoide(H);
return HOutput;
}
```

```
double calculateOutputDecisions(double w[2], double h0, double h1){
double outputSum = h0*w[0] + h1*w[1];
double output = fSigmoide(outputSum);
return output;
}
```

As we described in Section 5.1.6, each smart street light has to execute three tasks. Accordingly, we present below the main parts of the C++ code that the Arduino executes to attend to the tasks of collecting data, making decisions and enforcing actions:

- Collecting data:

```
void getInputs(){
lightSensor = readLightSensor();
motionSensor = readMotionSensor();
previousListeningDecision = listeningDecision;
if ( listeningDecision ==1){
receivedSignal = receiveWirelessData();
}
else
receivedSignal = 0;
}
```

- Making Decision (calculating output decisions based on code of the evolved neural network functions - see Section 5.1.6):

```

double weightsH0[4] = 1.2, -0.8, 1.6, -0.5;
double weightsH1[4] = 1.6, -0.8, 1.5, -0.3;
double H0 = calculateHiddenUnitOutput(weightsH0);
double H1 = calculateHiddenUnitOutput(weightsH1);

...

double weightsTransmitterOutput[2] = -0.6, -0.2 ;
double transmitterOutput = calculateOutputDecisions(weightsTransmitterOutput,
    H0, H1);

...

double weightslisteningDecision [2] = -0.9, -0.7;
double listeningDecisionOutput =
    calculateOutputDecisions(weightslisteningDecision, H0, H1);

...

double weightslightDecision [2] = 1.7, -0.4;
double lightDecisionOutput = calculateOutputDecisions(weightslightDecision, H0,
    H1);
if (lightDecisionOutput > threshold2){
    lightDecision = 1.0;
}
else {
    if (lightDecisionOutput > threshold1){
        lightDecision = 0.5;
    }
    else lightDecision = 0.0;
}

```

– Enforcing action:

```

void setOutputs(){
    ...
    sendWirelessData(transmitterSignal);
    ...
    writeLed(lightDecision);
    ...
}

void writeLed(double value){
    if (value == 1){
        digitalWrite(ledPin, HIGH);
        digitalWrite(led2Pin, HIGH);
    }
    else if (value == 0.5){
        digitalWrite(ledPin, HIGH);
        digitalWrite(led2Pin, LOW);
    }
}

```

```

}
else {
  digitalWrite(ledPin, LOW);
  digitalWrite(led2Pin, LOW);
}
}

```

Testing Physical Smart Street Lights in a Real Scenario

In a **controlled real scenario**, we put three prototypes of the smart street lights using the evolved neural network into operation. We distributed them in the scenario as shown in Figure 5.18. To compare the behavior of

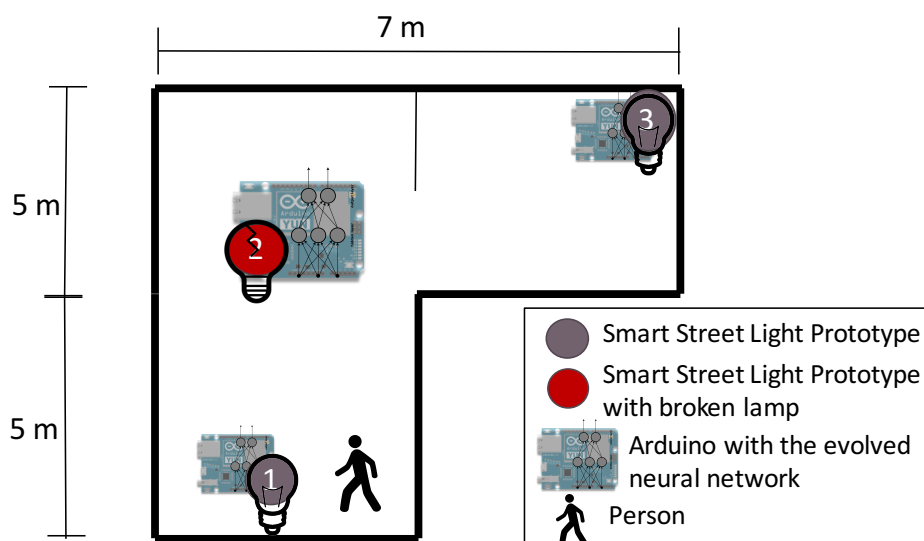


Figure 5.18: Real Scenario where we tested a network of three smart street lights prototypes.

physical smart street lights to the simulated ones, we also collected logs from the Arduinos¹. As we could observe, the behavior of the physical smart street lights was similar to the simulated ones: it switches lamps ON if it receives a signal different from 0.0 or detects the approximation of a person. However, we cannot assure that a street light is receiving the signal from the closest street light. In addition, different from the simulator, the real scenario is a distributed environment composed of asynchronous components with different clocks. But, as we are leading with a controlled environment with few resources, we cannot observe significant differences.

¹All files that were generated during the development of this work, such as genetic algorithm files, log program and arduino code, are available at <http://www.inf.puc-rio.br/~nnascimento/streetlight.html>

5.1.7

Discussion

We believe these preliminary results are promising. We proposed the use of the embodied agents concept to model smart things. To illustrate, we modeled and implemented smart fruit bowls and smart street lights. Each embodied agent had sensors and actuators to interact to the environment, and used an artificial neural network as an internal controller. In addition, we used a supervised learning algorithm to allow fruit bowls to learn about fruit spoilage process, and we used a genetic algorithm to allow street lights to self-develop their own behaviors through a non-supervised training. As a result of the second experiment, a group of initially non-communicating smart street lights developed a simple communication system. By communicating, the group of street lights seems to cooperate in order to achieve collective targets. For example, to maintain the maximum visual comfort in illuminated areas, the street lights used communication to reduce the impact of broken lamps.

We provided street lights with the possibility of disabling the feature of receiving signals from neighboring street lights. In an initial instance, we did not consider broken lamps. Therefore, as the action of communication increases energy consumption, the street lights decided to disable this feature. However, when we added broken lamps to the scenario, during the evolutionary process, the solution of enabling a communication system among street lights provided better results. Therefore, as shown in the rules generated by the evolved neural network, a smart street light takes lightSensor, motionSensor and wirelessReceived inputs into account to make decisions. Thus, the best solution does not ignore any of these parameters.

One advantage of engineering physical devices based on embodied cognition is that the found solution normally is sufficiently generic. To estimate how generic is the approach, we simulated another neighborhood with a different number of street lights and a different configuration map, then we applied this best solution to this new scenario. The results showed that the evolved street lights' behavior does not vary based on the number of street lights, and the lighting application continues functioning well even if we disable some street lights in the scenario.

5.1.8

How the proposed framework adheres to the reference model

Table 5.6 shows how this proposed framework adheres to the proposed reference model by concretizing the high-level statechart components. The

component “agent configuration” is developed for this approach as a reconfigurable system that contains the characteristics that can be used to compose the set of agents.

Table 5.6: Case I: Main statechart components.

Statechart Components		Approach
Agent Configuration	Body	A configuration file describing the sensors and actuators that will compose the agent’s body
	Controller	A configuration file specifying the neural network. In addition, agent’s controller is represented by an abstract class, making it possible to be adjusted during the system execution
Agent Behavior		An specific agent, named <i>AdaptiveAgent</i> , which acts based on the control loop and represents the embodied agent behavior. It reads data through the agent’s sensors, makes decisions through the neural network, and takes actions through the agent’s outputs.
Environment		An independent application that configures a dataset or scenario to be interacted with embodied agents
Task Evaluation		An specific agent, named <i>ObserverAgent</i> , which is responsible to evaluate the group of agents and to adjust the neural network that has been used by this group of embodied agents

5.2

An Architecture for Embodied Agents Reconfiguration

In this section, we propose a self-configurable embodied agent approach that allows to investigate how a collection of embodied agents can be configured to deal with the system’s requirements and environmental changes. To meet the system’s requirements (e.g. performance), this configuration consists of testing the behavior of agents by using different subsets of body, behavior and analysis architecture features. To deal with environmental changes, this configuration consists of evolving or adapting the set of agents by adding or removing features.

This investigation is associated to the following questions:

- RQ3. How to define an architecture to configure the body and controller of the agents based on the environment variability?
- RQ3.1. What is the variability related to embodied agents and how can this variability be represented?
- RQ3.2. How can we represent the effect of the relevant environment changes on the reconfiguration of the embodied agents?

The ability of a software system to be configured for different contexts and scenarios is called *variability* (Galster et al., 2014). According to Galster et al. (2014), achieving variability in software systems requires software engineers to adopt suitable methods and tools for representing, managing and reasoning about change. Therefore, before defining an architecture to configure agents based on the environment variability, we must investigate and represent which kind of variabilities are related to this kind of agent and to the environment in which it may be situated.

5.2.1

RQ3.1. What is the variability related to embodied agents and how can this variability be represented? - Variability in Embodied Agents

There are several options for physical components and software behaviors for the design of physical agents (Ayala et al., 2015). According to existing experiments (Soni and Kandasamy, 2017) and our experience with embodied agents (Nascimento and Lucena, 2017; Nascimento Marx Leles Viana, 2016; Nascimento and Lucena, 2017), we introduce possible variants of an embodied agent in (Nascimento et al., 2018). Our approach incorporates feature-oriented domain analysis (FODA) (Pohl et al., 2005) notation to explore this variability. According to the FODA notation, features can be classified as mandatory, optional and alternative. Alternative features are not to be used in the same instance.

In (Nascimento et al., 2018), we identified three main variation points to handle in order to create an embodied agent: (i) the body variability (i.e. number, types and brands of sensors and actuators); (ii) the complexity of the behavior of the agent, which varies based on the physical components that are operated by the agent (e.g. if this agent is able to communicate, the number of signals agents are able to exchange); and (iii) the agent's controller that allows the agent to sense the environment and behave accordingly (e.g. if this agent controller is a neural network, in terms of its architectural variability the type of activation function and the number of neurons should be considered).

Body and Behavior Variability

The physical devices may vary in terms of the types of sensors, such as temperature and humidity, and in terms of actuators, as depicted in the feature diagram presented in Figure 5.19. Each sensor can also vary in terms of brands, changing such parameters as energy consumption and battery life. Note that, depending on the application domain, this feature diagram may contain different and more specific features. For example, to create smart street light agents, we can provide a specific version of this feature model, discarding some options of sensors, such as heart, EEG and pressure sensors.

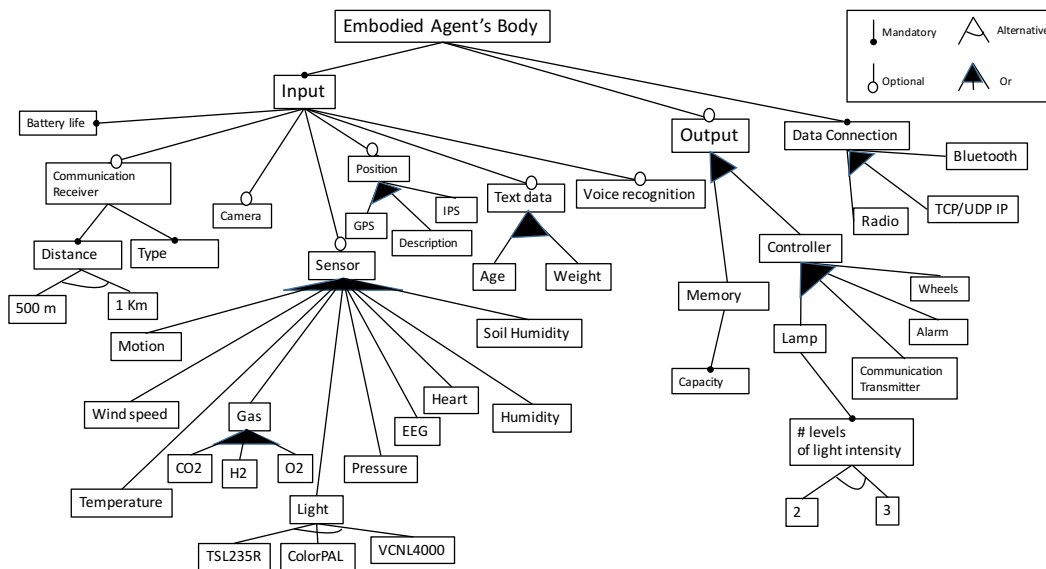


Figure 5.19: Feature model of embodied agents' body.

The complexity of the behavior of the agent will vary based on the physical components that are operated by the agent. For example, if an agent is able to activate an alarm, which kinds of alerts can this agent generate? If this agent is able to communicate, how many words is this agent able to communicate? If this agent is able to control the temperature of a room, what are the threshold values set to change the room's temperature?

Controller Variability

In addition, we also need to deal with variants in agent architecture that the agent uses to sense the environment and behave accordingly. For example, this controller's architecture can be a decision tree, a state machine or a neural network. Many approaches (Marocco and Nolfi, 2007; Nascimento and Lucena, 2017) use *neuroevolution*, which is "a learning algorithm which uses genetic algorithms to train neural networks" (Whiteson et al., 2005)). This type of network determines the behavior of an agent automatically

based on its physical characteristics and the environment being monitored. A neural network is a well-known approach to provide responses dynamically and automatically, and create a mapping of input-output relations, which may compactly represent a set of “if..then” conditions (Nascimento and Lucena, 2017), such as: “if the temperature is below 10°C, then turn on the heat.” However, finding an appropriate neural network architecture based on the physical features that were selected for an agent, is not easy. To model the neural network, we also need to account for its architectural variability, such as the activation function, the number of layers and neurons and properties such as the use of winner-take-all (WTA) as a neural selection mechanisms (Fukai and Tanaka, 1997) and the inclusion of recurrent connections (Marocco and Nolfi, 2007). We explore these variabilities in Figure 5.20.

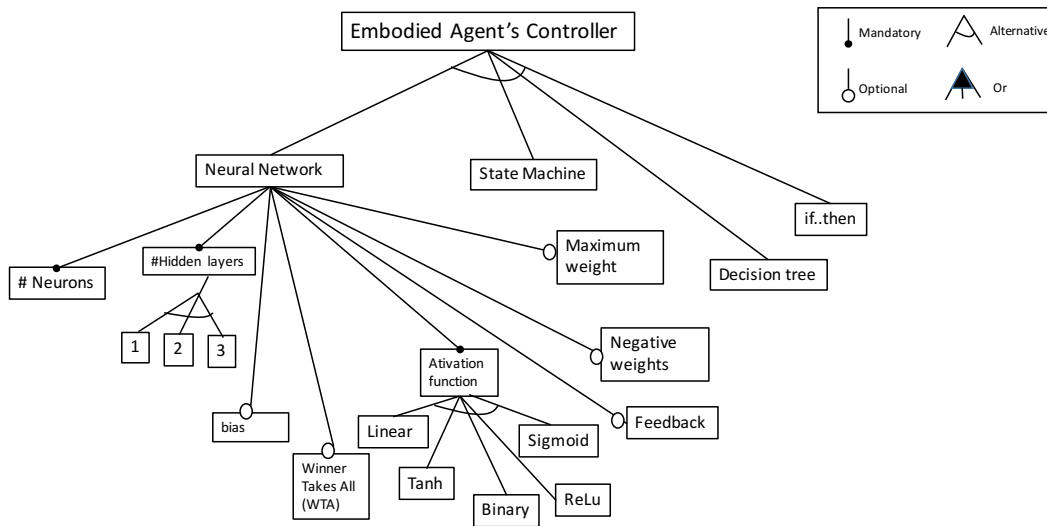


Figure 5.20: Feature model of an embodied agents' controller.

With respect to controller's variabilities, Marocco and Nolfi (2007), performed two experiments with the same embodied agents, varying only the neural network architectures and neural activation functions. In the first experiment, they used a neural network without internal neurons, while in the second experiment, they used a neural network with internal neurons and recurrent connections. In addition, they also used different functions to compute the neurons' outputs. Based only on the neural network characteristics, they classified the robots from the first experiment as reactive robots (i.e. “motor actions can only be determined on the basis of the current sensory state”), and non-reactive robots (i.e. “motor actions are also influenced by previous sensory and internal states”). Marocco and Nolfi (2007) analyzed whether the type of neural architecture influenced the performance of a team of robots. They showed that the differences in performance between reactive and non-

reactive robots vary according to the environmental conditions and how the robots have been evaluated.

(Oliveira and Loula, 2014) investigated symbol representations in communication based on the neural architecture topology that is used to control an embodied agent. They found that the communication system varies according to how the hidden layers connect the visual inputs to the auditory inputs.

(Jarraya et al., 2018) propose a multiagent approach for pervasive computing that aims to identify human activities in smart homes. In their approach, the set of agents must observe sensor data and make local predictions. Jarraya et al. (2018) state that “depending on the nature of sensor data, agents may hold different types of classifiers.”

These findings have helped us to conclude that to support the design of embodied agents, we need to account for the variability of the physical body and the architecture that analyses the inputs. In addition, we also concluded that the agent’s controller cannot be considered as a black box in the system, since its structure must fit the characteristics that were selected to compose the body and behavior of the agent.

The summary of variabilities captured by other approaches and by the approach proposed in this thesis is shown in Table 5.7.

5.2.2
Description of the Architecture

Based on the two main variation points we have identified, we propose a platform to design self-configurable embodied agents applications. Figure 5.21 depicts the high-level model of our proposed approach.

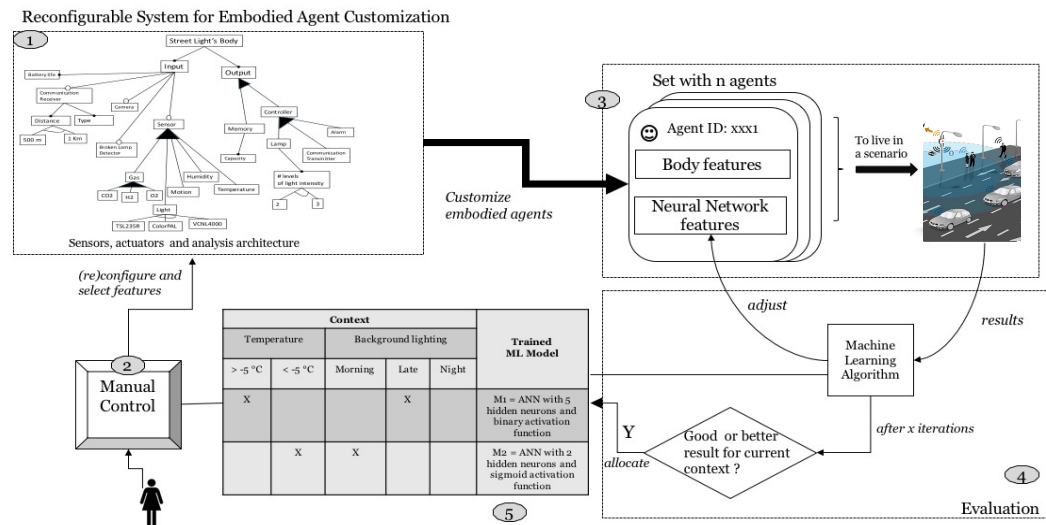


Figure 5.21: High-level model of the self-configurable agent approach to generate embodied agents.

Table 5.7: Summary of embodied agents variability.

Body Variability		Behavior Var.	Controller Var.
Number of sensors		Number and type of communication signals	Number of layers
Type of sensors (e.g. temperature, humidity, motion, lighting, gases)		Notification types (e.g. alerts)	Number of neurons per layer
Sensors	Calibration of sensors (e.g. temperature detector range, range of presence detection, reaction time, range of colors detection)	Thresholds to activate notifications	Activation Function (e.g. linear, sigmoid)
	Energy Consumption	Classification (e.g. air quality, disease)	Properties (e.g. WTA, feedback)
	Battery life	Prediction (e.g. heart attack, fruit crop yield)	Architecture (e.g.full connected, output layer connected to all of the hidden units)
	Communication device		
Range of communication devices (e.g. short range, long range)			
Number and type of motors			
Number and type of actuators (e.g. alarm)			

Basically, this platform contains five modules: i) a reconfigurable system that contains the characteristics that can be used to compose the set of agents, according to the application domain; ii) a manual control that allows a domain expert to select the first set of features manually; iii) the creation of a set of agents containing the selected characteristics that are also able to use a neural network to learn about the environment; iv) a module for evaluating feedback tasks, by investigating the performance of the group of agents in the application scenario during the learning execution. The evaluation process has to be implemented according to the application and the learning algorithm. For example, if an application for automobile traffic control has the goal of reducing urban traffic congestion, the evaluation may be performed based on the number of vehicles that had finished their routes in a specific period; and (v) a module to store and retrieve machine learning models based on the context, as detailed in Section 5.2.2. It allows the set of agents to switch the analysis architecture according to the context at runtime. The domain expert

can also use this context information to reconfigure the set of agents.

RQ3.2. How can we represent the effect of the relevant environment changes on the reconfiguration of the embodied agents?

To represent the effect of the relevant environment changes on the reconfiguration of the agents, the first step is to identify the attributes of the environment that we need to consider in contextualizing the information and to determine the contexts that this application will consider. For example, if we have an application that contains a set of robots that interact with a place, this place can be characterized by the temperature and the background lighting information. Thus, the different contexts for this application will only vary the temperature and background lighting information. If we have an application that uses a glove for hand-gesture recognition, we may consider the hand shape information as a contextual information in evaluating different models. In an application for self-tracking, which monitors a person’s data and performs disease prediction, a person can be distinguished based on the country of residence. The information that we select to characterize one or more entities in the application will be used as a parameter to detect changes, to separate the input space, to train and to deploy the ML models.

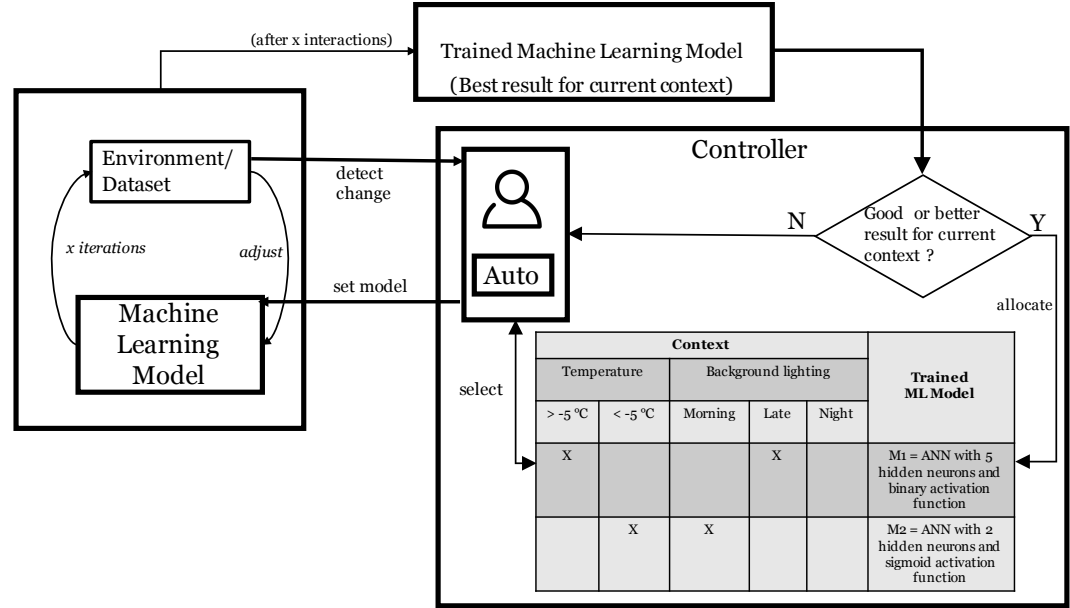


Figure 5.22: Schematic illustration of our architecture to store and retrieve machine learning models based on context. Published in (Nascimento et al., 2018).

This module to store and retrieve machine learning models based on the context consists of two components, as depicted in Figure 5.22: (i) a system that uses a machine learning model to analyze a dataset or interact

with an environment while a learning algorithm tries to maximize a given scoring function; and (ii) a control module that configures machine learning models, monitors the context changes, and maintains a history of the trained ML-based models that provide the best result for each one of the operating contexts. When the context changes, the quality of the results achieved by the ML model that is in current use may decrease. As a consequence, if context changes are known, the controller will revert to a previous configuration instead of retraining itself. Otherwise, it will test new machine learning models. To allow the system to find a new model, our architecture supports manual and automatic controllers to manage permissible run-time adaptations caused by environmental or system changes.

In short, to illustrate the use of our approach, consider a smart solar battery in an aircraft that uses a trained artificial neural network (ANN) to make decisions. In this illustrative example, by using a temporal context, we can evaluate if the results can be improved if the model that we use in the winter is different from the one in the summer. For example, suppose that during the training step, we observed that during the summer, a neural network with binary activation function outperformed a neural network with sigmoid function, but during the winter, ANN with sigmoid was better. Then, based on our proposed architecture, the solar battery will be deployed with two models: one for the summer and another for the winter. It is an important approach for this kind of application, where accuracy is the most important factor.

Modeling Embodied Agents

As we described in Section 5.1, FIoT does not support agents' body reconfiguration, but only their controller adjustment. To support body and controller reconfiguration, as described in our proposed statechart models, we adapted FIoT's model to our new approach. Accordingly, Figure 5.23 depicts the class diagram of an embodied agent. Each embodied agent contains a controller that is a result of the composition of a subset of the body features set and a subset of the controller features set.

To create a system that can change how the embodied agents sense the environment at runtime, without actually changing the body of the agents, we added the physical characteristics to the controller. For example, if we deploy an agent with five sensors, we can test how this agent behaves while using controllers with different physical configurations. Even when we deploy the agent with a controller that considers all input sensors, we can change it at runtime to another controller that uses only part of these sensors.

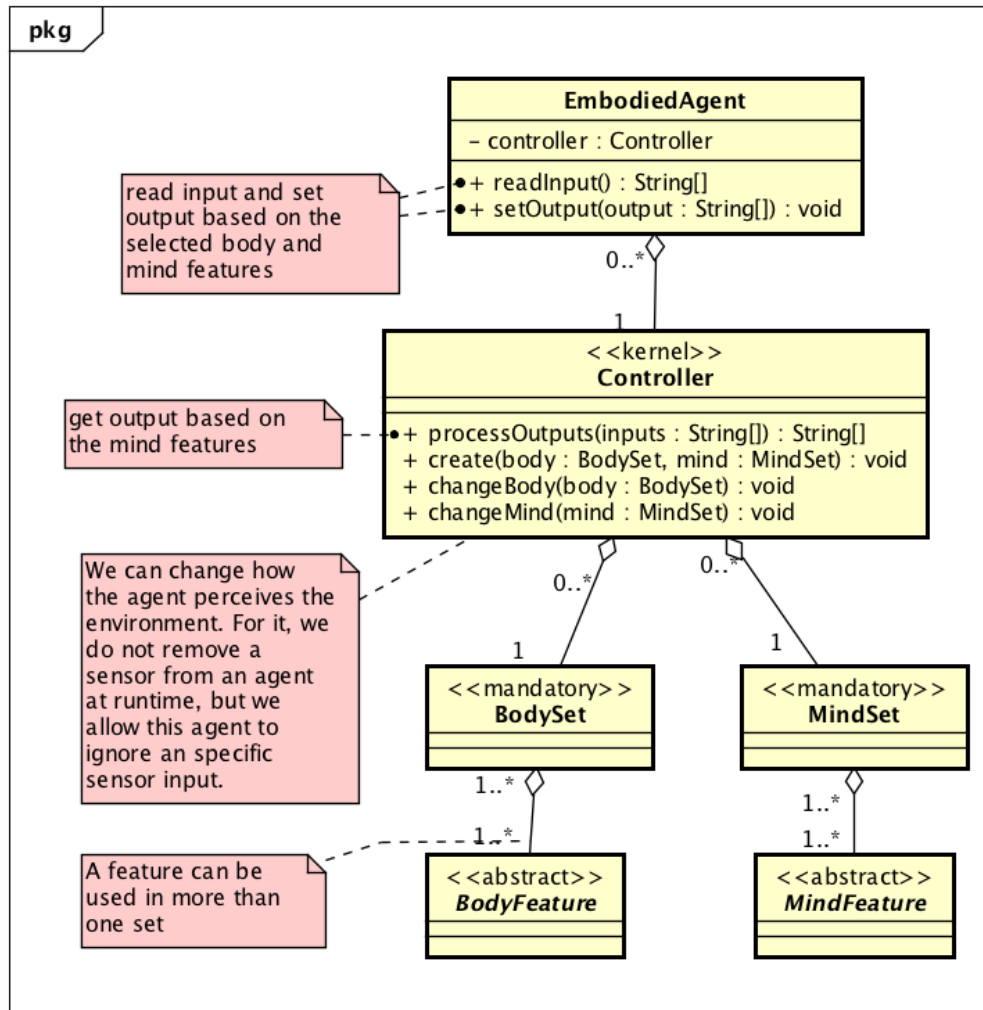


Figure 5.23: Modeling variability-aware embodied agents.

An agent to execute Task Evaluation actions

Following the idea described in FIoT, we have a group of agents using the neural controller to take decisions, while there is an agent, named Observer-Agent, evaluating this group of agents and optimizing this neural controller accordingly. Previously, this agent could change only the neural weights. Now, we extended the ObserverAgent in order to allow it to change at runtime the whole ANN's structure. Therefore, during an experiment, a collection of embodied agents can start the experiment using a neural network with three sensors, two hidden layers and a hyperbolic tangent function as the transfer function of all neurons; and finish this same experiment using a totally different neural network, with one sensor, one hidden layer and a sigmoid function.

In our new approach, we called this agent LearningControlAgent, as depicted in Figure 5.24. For instance, the engineer can provide two or more different ANNS. For example, he/she can generate three neural networks

varying only the activation function and the topology. The engineer can explore all possible configurations of the feature model, generating different options of ANNs. The LearningControlAgent has the list of controllers that were generated by the user.

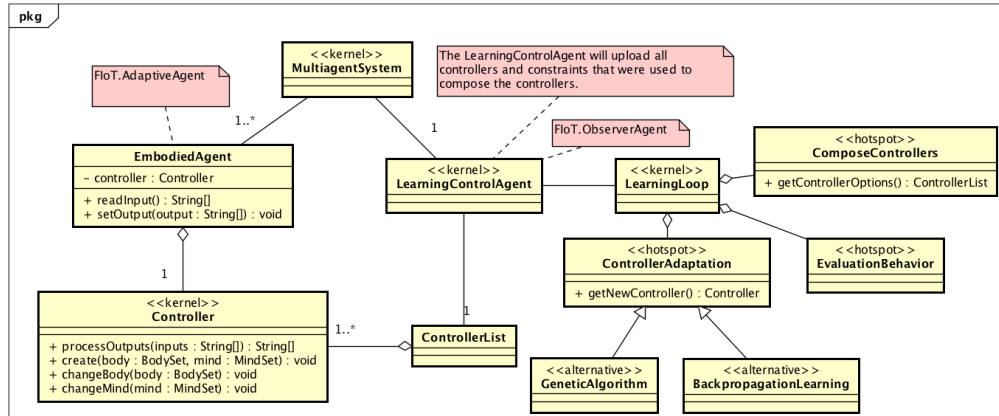


Figure 5.24: Class diagram of a multiagent system composed of IoT embodied agents.

5.2.3

Evaluation: Example of Application

To give more details about and illustrate the use of our proposed approach, we selected one example from the IoT domain: a smart street light application. In this application scenario, we consider a set of street lights distributed in a neighborhood.

5.2.4

Application I: Reconfigurable Smart Street Lights

In the Section 5.1.6, we described a set of street lights, which are distributed in a simulated neighborhood, with neural networks in order to enable them to make decisions based on the data collected from the environment. They used a genetic algorithm to train the neural networks.

Each street light operates in an environment in which the background light can be bright or dark. With respect to the environmental background light, the application scenario has some variants: (i) night (background light is equal to 0.0); (ii) late afternoon (background light is equal to 0.5); and (iii) morning (background light is equal to 1.0). Each street light contains a lighting sensor, but the local brightness also interferes with the sensor measurement.

Problem

During a first simulation, while the background light was always bright, the collection of street lights found a solution that provided a performance, say $X+1$, during the morning. After the environment changed to the night, the lights' solution was adjusted to deal with this change. However, this new generic solution presents a performance, say X , during the morning, and the street light is unable to return to its previous configuration, as the street light does not maintain different versions of its configuration. This configuration history could enable the street light to maximize its performance during different parts of the day.

In this scenario, is it better to create a street light with a general analysis architecture that can deal with all variants of the background lighting; or a street light that is able to switch its analysis architecture to specialized solutions that were selected for each of the background lighting variants?

Experimental Setting: Implementing a context-aware-based decision

We simulated four scenarios, varying only the background light configuration: (i) always bright; (ii) always dark; (iii) always late afternoon; and (iv) one that dynamically changes the background light at runtime (i.e. some-times the background light is bright and at other periods of time, such as at night, the background light is dark).

As described in Section 5.1.6, we provided each street light with a neural network to make decisions and we used a genetic algorithm to train the neural network. However, different from the presentation in Section 5.1.6, where one unique neural model dealt with all background light variants, we deployed the simulated street lights with more than one model, which varied according to the background light context. Further, we compared the set of street lights that can use more than one model at runtime against the set of street lights that uses only a single generic model. To generate these models, we trained different neural networks in different contexts, as depicted in Figure 5.25.

First, we trained a neural network, which consists of five hidden units and uses a binary activation function, putting the set of street lights to interact with the first scenario (always bright). Then, we trained a sigmoid neural network three times (using the scenarios (ii), (iii) and (iv), separately), generating three trained models with particular weight values. In total, we saved four models, such as that one that was generated while the set of street lights interacted only with the first scenario.

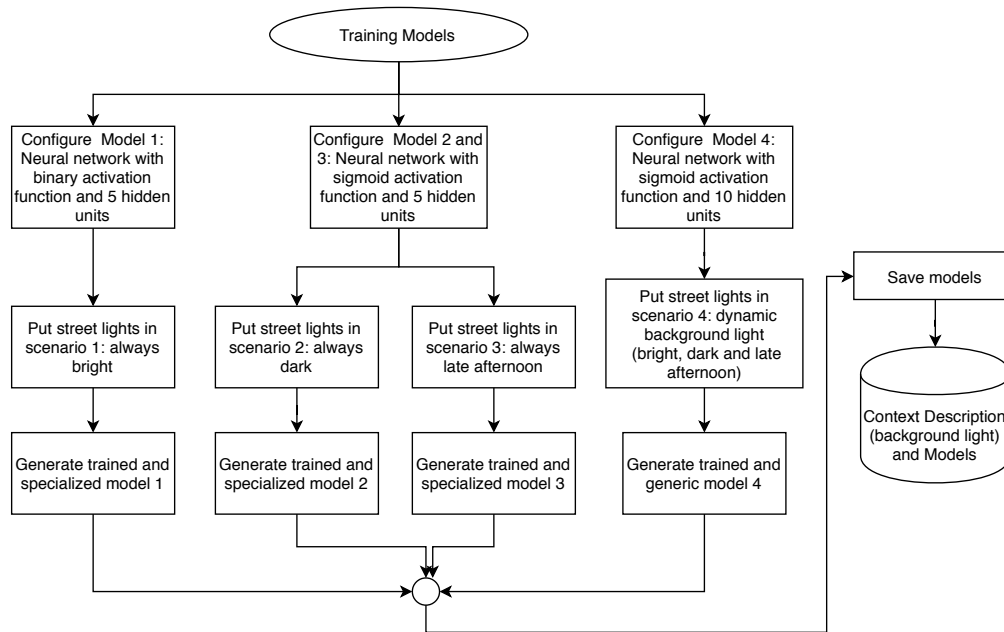


Figure 5.25: Training step: varying the background light context in a street light scenario to generate generic and specialized ML models.

Experimental Results: Evaluating Street Lights Performance

As we described, we generated four models by training the neural networks in scenarios with different background lighting contexts: three specific models (one for each background lighting configuration) and one general model (that was trained while the set of street lights interacted with dynamic background lighting). Table 5.8 shows the performance results that were achieved by the set of street lights while using these models in different contexts. For more details concerning the performance calculations, see Section 5.1.6.

According to Table 5.8, the models achieved different results for each part of the day. For instance, the generic model provided the best results for the night and morning. However, the specific model outperformed the generic model during the late afternoon. As the general model is dealing with a more complex scenario, we increased the number of interactions from 20 to 100 to verify if its results could be improved, but they could not.

Accordingly, if we train a unique model to deal with all background lighting variants, as presented in Section 5.1.6, the average performance of the set of street lights in this application scenario is $\approx 59.39\%$. If the set of street lights is able to select the most appropriate model for each one of the background light configurations, the average performance is $\approx 60.31\%$. Probably, this difference would be increased if we considered more context parameters.

Table 5.8: Street lights’ performance results obtained while executing different models with different contexts.

ML-based Model		# Interactions	Background Lighting		
			0.0	0.5	1.0
Specific Models	0.0	20	44.85%	x	x
	0.5	20	x	65.49%	x
	1.0	20	x	x	70%
Generalist Model		20	45.42%	62.74%	70%

Therefore, our results show that the best solution for this application scenario is to create a system that is able to use the generic model at night and in the morning, and to switch to the specific model in the late afternoon, as depicted in Figure 5.26.

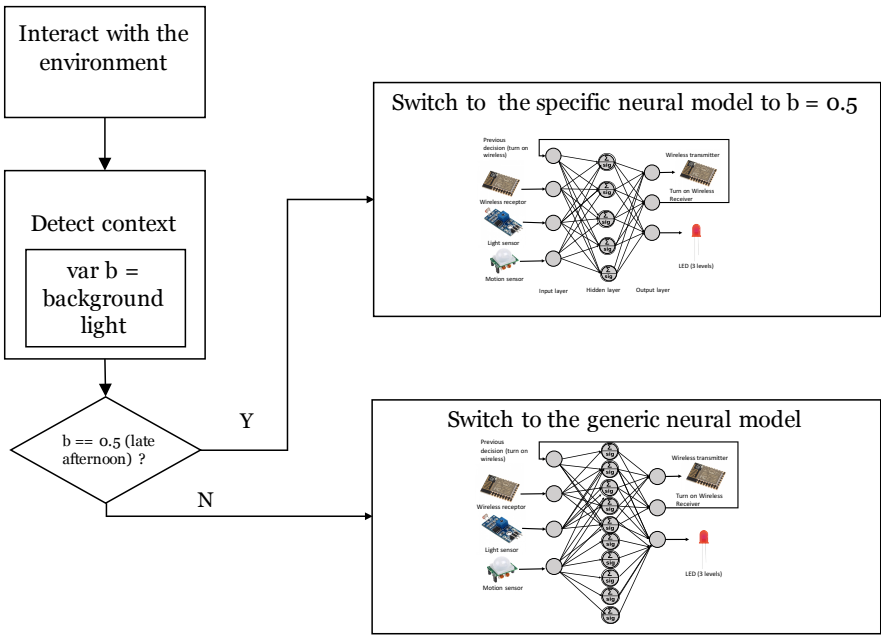


Figure 5.26: Deployment step: selecting the trained model to use according to the context.

However, the difference between the approaches’ average performance is less than 1%. Thus, the software engineer must evaluate which approach fits the application requirements better. For example, if the application is critical and performance is the most important requirement, our proposed approach should be considered. If the storage capacity of the application is limited or the system cannot connect to ML web services, finding a general good model may be the most appropriate approach.

5.2.5

How the proposed approach adheres to the reference model

Table 5.9 shows how this proposed architecture adheres to the proposed reference model by concretizing the high-level statechart components. The component “agent configuration” is developed for this approach as a reconfigurable system that contains the characteristics that can be used to compose the set of agents.

Table 5.9: Case II: Main statechart components.

Statechart Components		Approach
Agent Configuration	Body	A feature model to represent embodied agents’ body variability.
	Controller	A feature model to represent embodied agents’ neural network variability.
Agent Behavior		<p>The creation of a set of agents containing the selected characteristics that are also able to use a neural network to learn about the environment.</p> <p>Agent senses and acts based on the selected body features. It decides based on the trained neural network’s configuration.</p>
Environment		The user needs to previously identify the attributes of the scenario or dataset that are needed to consider in contextualizing the information.
Task Evaluation		A module for evaluating feedback tasks, by investigating the performance of the group of agents in the application scenario. In addition, a module to store and retrieve neural networks based on the context.

5.3

An Approach to Test Embodied Agents

According to (Bredeche et al., 2018), there is a gap in the literature regarding the testing of embodied agents. A further complication is that current embodied agents-based approaches may involve different characteristics, such as different physical configurations, asynchronous, and a learning-based behavior, as we described in Section 4. In addition, the environment in which these agents are situated also may have many variants.

Therefore, in order to answer the following question “**RQ4. How to design and implement an approach to test embodied agents and their variability?**”, this section describes an approach that we propose to test embodied agents and their variability.

In (Nascimento et al., 2017), we presented a preliminary version of a publish-subscribe-based architecture that was implemented² to make feasible the development of multi-level tests based on logging for multiagent systems. By using this platform, it is possible to test the behavior of individual agents and the behavior of a group of agents. However, we only showed the usability of our platform by testing a very simple MAS application - a marketplace to buy and sell books on-line. Then, we improved this architecture and presented a new approach in (Nascimento et al., 2019) that makes it possible to diagnose failures in a more complex MAS application, one that involves embodied agents.

To illustrate and evaluate the use of the proposed approach, we used the application presented in Section 5.1.6, which we developed by using the “Framework for the Internet of Things” (FIoT) (Nascimento and Lucena, 2017) (see Section 5.1);

5.3.1 Description of the Testing Approach

To test applications based on embodied agents, our approach promotes the development of tests separated into six perspectives: (i) a designing perspective (i.e. a test application to evaluate the sensors, actuators and analysis architecture that were selected to compose the agent); (ii) a learning perspective (i.e. a test application to inspect the interactions generated because of the learning algorithm that optimizes agent’s controller); (iii) a scenario perspective (i.e. a test application to consume the logs generated by the application scenario); (iv) a task evaluation perspective (i.e. a test application to evaluate whether the application goal was achieved); (v) a behavior perspective (i.e. considering the tasks that an individual embodied agent must be able to execute according to its body and controller); and (vi) a framework perspective (i.e. evaluating the agent interactions generated because of the framework that we used to create the application).

We need to customize these tests according to the application. In general, at the task evaluation level, we should verify if the system is able to solve the problem for which it is conceived. For example, our streetlight application has

²The source of the test system is available at <http://www.inf.puc-rio.br/~nascimento/MAS-tests.html>

the goal of achieving an specific energy consumption target and maintaining the maximum visual comfort in illuminated areas in order to enable people to finish their routes. If the group of embodied agents does not solve this problem, we should investigate the other perspective tasks to understand why the process failed.

Design and Implementation: An Architecture based on Metadata and the Publish-Subscribe Paradigm

We developed a publish-subscribe-based architecture as a foundation for developing different kinds of test applications for MASs at different perspectives. Our goal is to provide mechanisms to capture and process logs generated by agents automatically. As depicted in Figure 5.27, their architecture consists of three layers: MAS Application (L1), Publish-Subscribe Communication (L2), and Test Applications (L3). The Publish-Subscribe Communication layer uses the RabbitMQ platform (see subsection 5.3.1) for delivering logs from agents (publishers) to be consumed by test applications (subscribers).

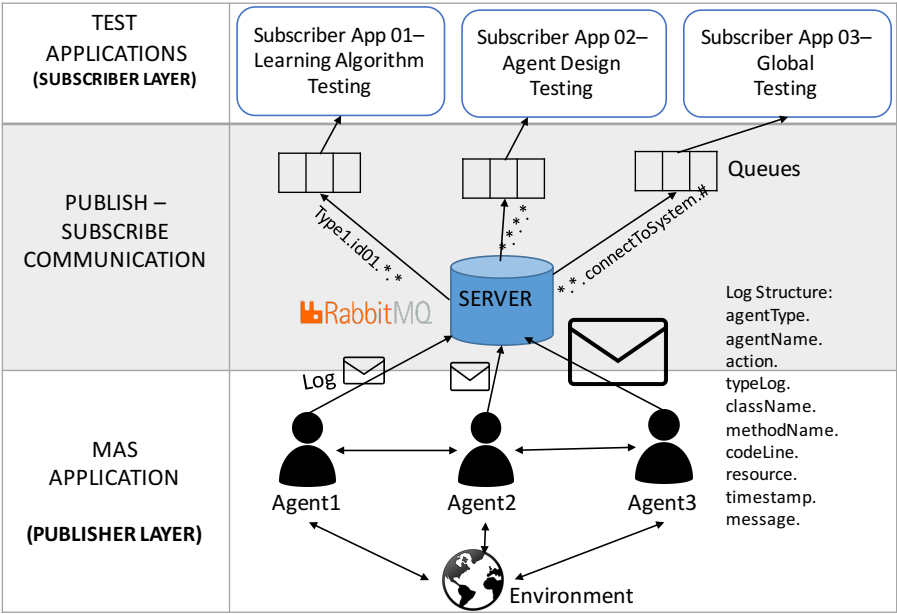


Figure 5.27: A Publish-Subscribe-based architecture to test MASs.

Each agent publishes logs with annotations that are composed of the following tags:

- *agentType*: the type of the agent (e.g OBSERVER, STREETLIGHT). In JADE, it refers to the name of the container where this agent lives;
- *agentName*: the name provided for the agent by the system developer/user (e.g streetlight01, streetlight02, observer01);

- *action*: the event that caused the log generation (e.g readMotionSensor, selectBestIndividuals, switchStreetLight);
- *typeLog*: types of logs (e.g error, info, warning);
- *className*, *methodName*, *codeLine*: necessary information to identify which parts of the code generated the event;
- *resource*: the main resource that has been manipulated or requested by an agent during an event execution (e.g neuralController, streetlight01Info, memory). It may be used to investigate all events that are related to a specific resource;
- *timestamp*: time that the log was created. It is used to sort all events into a single timeline (Araújo and Staa, 2014);
- *message*: a description of the event.

Thus, a log message must meet the pattern “(agent-Type).(agentName).(action).(typeLog).(className).(methodName).

(codeLine).(resource).(timestamp).(message).” Each application will have a set of values that each tag may assume, except the message tag is an open field.

All agents in the MAS application layer are also a TestableAgent type. As shown in Figure 5.28, a Testable agent extends the JADE agent. Thus, it complies with FIPA specifications. A Testable agent uses the RabbitMQ properties to send logs with annotations as messages.

These logs can be published from any part of the agent’s code. Via the TestableAgent class and JADE properties, some tags have their values attributed autonomously, such as agentType, agentName and timestamp.

```
public abstract class TestableAgent extends Agent {
    public abstract String getNameClass();
    public abstract boolean getTestMode();
    public void sendLog(LogValues.Action action, LogValues.TypeLog typeLog,
        LogValues.MethodName methodName,
        String codeLine, LogValues.Resource resource, String message) {...5 lines }
    private void send(LogContext newLog) {
        if(this.getTestMode() ==true){
            try {
                String dateInFormat = getDateInFormat(newLog);
                String log[] = {newLog.getAgentType()+ "." + newLog.getAgentName() + "." +
                    newLog.getAction() + "." +
                    newLog.getTypeLog() + "." + newLog.getClassName()+ "." +
                    newLog.getMethodName() + "." + newLog.getCodeLine()+ "." +
                    newLog.getResource() + "." + dateInFormat + "." + newLog.getMessage()};
                RabbitMQ.sendMessage(log);
            } catch (Exception ex) {
                Logger.getLogger(TestableAgent.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

Figure 5.28: Testable Agent class.

The RabbitMQ autonomously delivers log messages to queues according to their tags’ values. As shown in Figure 5.27, each test application defines a

binding key in order to subscribe itself to consume messages from a specific queue. For example, a test application that monitors only error logs from the Observer agent must have the binding key “Observer.*.*.error.#.” Therefore, this application will consume any log with the tuples (agentType,Observer) and (typeLog,error). It is also possible to create applications that use multiple bindings. For example, if a performance test application needs to calculate the number of Adaptive agents that are connected to the system, this application will have to consume logs with different action values. Thus, it needs to consume logs with the tuples (action,connectToSystem) and (action,beDestroyed).

Test applications do not interfere on the execution of each other. Each test class extends the class `RabbitMQConsumer` that starts an independent process to consume messages from a specific queue. By using queues, the publisher generates a set of information elements without the need of knowing which applications will consume them. In addition, more than one application can consume the same data, but giving them different treatments. To understand more about the characteristics of RabbitMQ that we used in our approach, see <https://www.rabbitmq.com/tutorials/tutorial-five-java.html> (Accessed in 03/2019).

RabbitMQ: Publish-Subscribe Platform

RabbitMQ (RabbitMQ, 2016) is a message-oriented middleware, which generates asynchronous, decoupling applications by separating sending and receiving data through a client and scalable server architecture. It can be easily integrated into an application to operate as a common platform to send and receive messages, maintaining messages in a safe place to live until received. RabbitMQ is a multi-platform that may be deployed in Java, C, Python, and many other programming languages. It can also be deployed in a cloud infrastructure.

By using RabbitMQ, it is possible to build a logging system based on the publish-subscribe architecture. The publisher is able to distribute log messages to many receivers, while the consumers have the possibility of selectively receiving the logs. Publisher and consumers communicate through queues. Each queue has a particular routing key that is a list of words, delimited by dots. There can be as many words in the routing key as you like, up to the limit of 255 bytes. These words can be anything, but usually they specify some features connected to the message. For example, if a developer specifies that a log message must meet the pattern “(month).(day).(deviceId).(typeLog)”, the valid routing keys would be “november.11.device01.error” and “november.15.device01.info” (RabbitMQ, 2016).

Therefore, a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key. However, there are two important special cases for binding keys (RabbitMQ, 2016):

- * (*star*) can substitute for exactly one word; and
- # (*hash*) can substitute for zero or more words.

Adapting FIoT Agents to be Testable Agents

```
public abstract class FIoTAgent extends TestableAgent {

public class ManagerAgent extends FIoTAgent {
    @Override
    public String getNameClass() {
        return "ManagerAgent";
    }

    @Override
    public boolean getTestMode() {
        return true;
    }
}

public class ObserverAgent extends FIoTAgent{
    @Override
    public String getNameClass() {
        return "ObserverAgent";
    }

    @Override
    public boolean getTestMode() {
        return true;
    }
}

public class AdaptiveAgent extends FIoTAgent {
    @Override
    public String getNameClass() {
        return "AdaptiveAgent";
    }

    @Override
    public boolean getTestMode() {
        return true;
    }
}
```

Figure 5.29: Making FIoT’s agents as Testable Agents.



The figure shows an IDE window with a project explorer on the left and a code editor on the right. The project explorer shows a package structure for 'FIoT' with sub-packages like 'fiot.agents', 'fiot.agents.behaviors', etc. The 'fiot.agents.logs' package is highlighted, containing 'LogAdaptiveAgent.java', 'LogManagerAgent.java', and 'LogObserverAgent.java'. The code editor shows the implementation of 'LogAdaptiveAgent' which implements 'LogValues' and defines 'Action', 'TypeLog', 'Resource', and 'MethodName' enums with their respective methods.

```
public class LogAdaptiveAgent implements LogValues{
    public enum Action implements LogValues.Action {
        connect, waitMsgFromSmartThing, receiveMsgFromSmartThing,
        receiveInputDataFromSmartThing, getController, useControllerToGetOutput,
        sendOutputToSmartThing;
    }
    public enum TypeLog implements LogValues.TypeLog{
        INFO, WARNING, ERROR;
    }
    public enum Resource implements LogValues.Resource{
        controller, smartThing;
    }
    public enum MethodName implements LogValues.MethodName{
        setup, create, setControlLoop, setInput, getOutput, controlLoop, read,
        readInputDevice;
    }
}

public enum Action implements LogValues.Action {
    connect, chooseAdaptationMethod, changeControllerConfiguration,
    startExecutionWithControllerConfiguration,
    startExecutionWithoutAdaptation,
    readSimulationResults,
    calculateEnergy, calculatePeople, calculateTripDuration, calculateFitness,
    achieveEnergyTarget, achievePeopleTarget,
    receiveMsgFromSmartThing, sendMsgToSmartThing,
    startGeneticAlgorithm, generateFirstPopulation, startNewGeneration,
    selectBestFitIndividuals, finishGeneticAlgorithm;
}

public enum TypeLog implements LogValues.TypeLog{
    INFO, WARNING, ERROR;
}

public enum Resource implements LogValues.Resource{
    solution, individual, generation, dataset, agent, smartthing;
}

public enum MethodName implements LogValues.MethodName{
    setup, create, observerLoop, executingNeatControl, trainingNeatControl,
    readResultSimulation, answerMessage, makeCopyGeneration;
}
```

Figure 5.30: Setting log values for each Testable FIoT agent.

Our first step was to allow FIoT agents to publish logs during the application execution, extending the TestableAgent class, as shown in Figure 5.29.

Then, we set the log values that can be published by each agent type. For example, the AdaptiveAgent can use the word ‘receiveInputDataFromSmartThing’ to replace the tag action in the annotated log, while the ObserverAgent can use ‘startGeneticAlgorithm’.

5.3.2

Evaluation: Example of Application

To give more details about and illustrate the use of our proposed approach, we selected the example of the smart street light application to develop a set of tests.

5.3.3

Application I: Testing Smart Street Lights

In short, this experiment involves developing reconfigurable streetlights. The overall goal of this application is to reduce the energy consumption while maintaining appropriate visibility in illuminated areas (see Section 5.1.6). For this purpose, each streetlight was provided with ambient brightness and motion sensors, and an actuator to control light intensity. In addition, they are able to interact with each other through a wireless communicator.

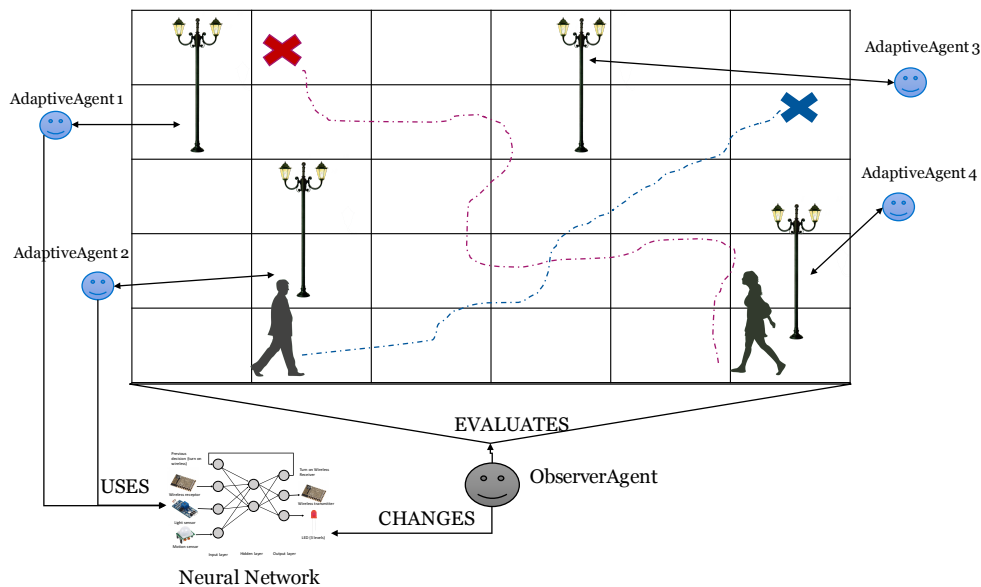


Figure 5.31: Overview of the general application architecture.

Each street light is controlled by an AdaptiveAgent, as shown in Figure 5.31. We used a neuroevolutionary algorithm to support the design of the street behaviors of the street lights automatically. Each streetlight uses a neural network to determine the communicating signals, and whether it turns on its lights. An ObserverAgent evaluates the overall application performance and

uses a genetic algorithm to optimize the AdaptiveAgents' neural network (i.e. adjusting agents' controller, as we describe in the statechart 4.5). As detailed in Section 5.1.6, this evaluation is based on energy consumption, the number of people that finished their routes before the simulation ends, and the total time spent by people moving during their trip:

$$fitness = (1.0 \times pPeople) - (0.6 \times pTrip) - (0.4 \times pEnergy) \quad (5-10)$$

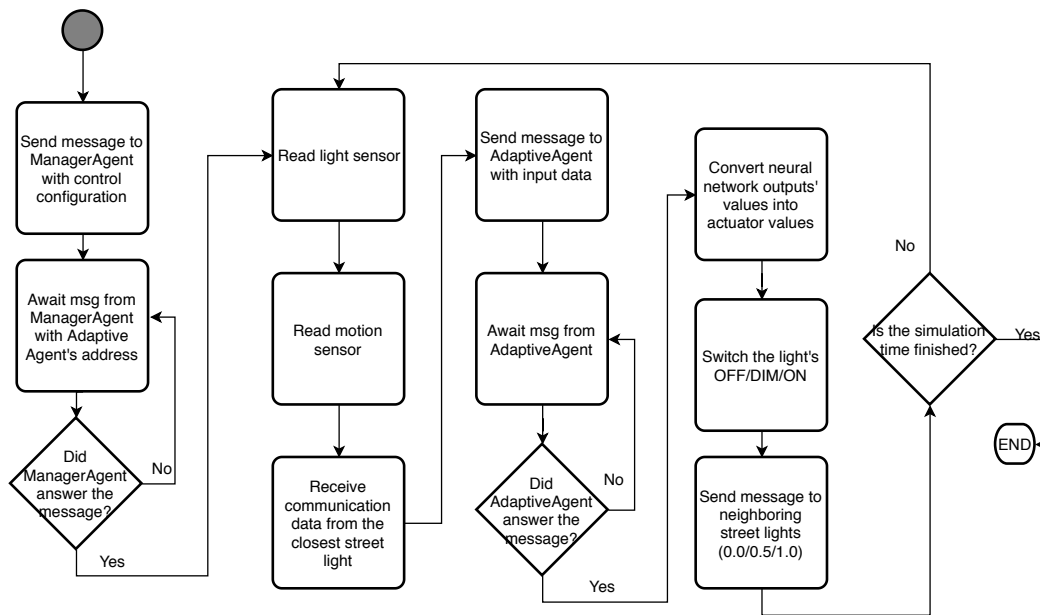


Figure 5.32: Activity diagram of the streetlights.

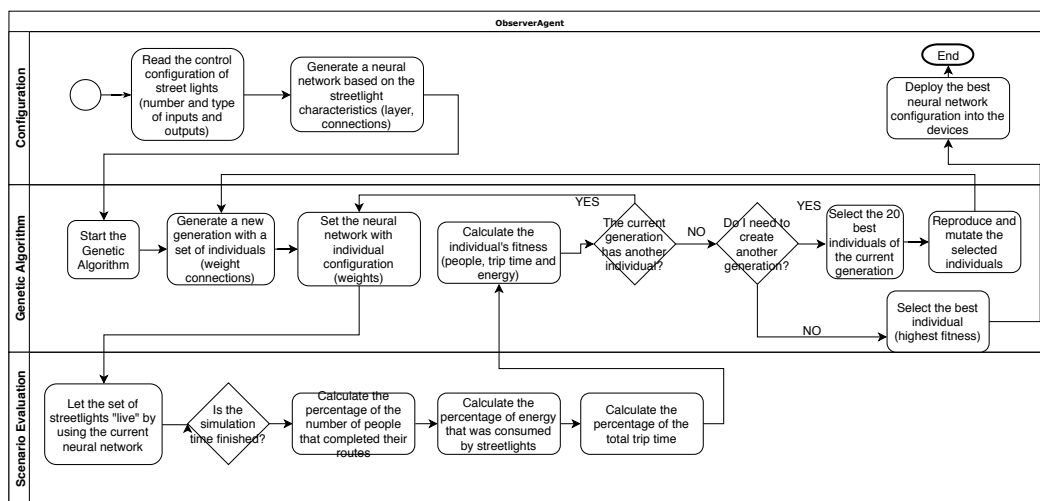


Figure 5.33: Activity diagram of the ObserverAgent.

In order to identify the functional tests, we first created activity diagrams for the street light agents and for the ObserverAgent, as depicted in Figures 5.32 and 5.33.

Experimental Setting

In our illustrative example, we can investigate the failures generated by the tasks associated with the framework (i.e. the ManagerAgent cannot identify new streetlights at the scenario), to the agent design (i.e. streetlight agents must detect people, but they do not have motion sensors), tasks related to the application scenario (i.e. streetlights should communicate, but the distance between them is higher than the wireless range), or the tasks related to the learning algorithm execution (i.e. the ObserverAgent is executing the genetic algorithm wrongly, selecting the worst solutions to compose a new generation instead of the best solutions).

The task evaluation perspective takes the global tasks into account, such as verifying whether the agents behavior guarantees that people finish their routes before the simulation ends and whether the system achieves a pre-specified energy consumption target. The behavior perspective considers the tasks that an individual embodied agent in the collection of streetlight agents must execute, such as collecting data, switching the light and communicating with the other agents.

By using our proposed architecture, we created some test applications to execute functional tests at some levels. In this experiment, we have one test application consuming logs related to the task evaluation perspective, monitoring the ObserverAgent and its learning algorithm execution, and two test apps related to the designing, scenario and behavior perspectives, monitoring the streetlight agents and their interaction with the environment. Thus, this section presents part of the test plan that we created and performed for testing this application.

Test Cases

We executed various test cases, taking six parameters into account: (i) perspective (e.g. related to agent design, the agent behavior, learning or scenario requirements); (ii) function (e.g. composed of a set of actions; for example, the function evaluateSolution may be composed of the actions calculateEnergy and calculateNumberPeople); (iii) procedure (e.g. a general description of the test); (iv) input (e.g. a resource, a component); (v) expected value (e.g. the result that will be produced when executing the test if the program satisfies its intended behavior); and (vi) validation method (e.g.

the strategies that a tester performs to evaluate the system, comparing the program execution against expected results).

Experimental Results: Testing Street Lights

Each test case execution produced several logs with meta-information annotations, which were consumed by test applications. Then, we used these logs as a validation method, as shown in Table 5.10. To validate a test case, the test application must verify whether the logs are appearing in the order described in the Validation Method column. Therefore, after the developer informs the logs from the validation column, the test application will automatically create a state machine, where each state represents an action. For example, Figures 5.34 and 5.35 illustrate the state machine that were created to validate the execution of the global test “evaluate solution” and the local test “switch the light ON”, respectively. As shown, the verification program defines the transition between states as a log. A transition will only occur when the expected log appears. Each state has a maximum wait time for the expected log(s). Thus, if the maximum wait time exceeds a threshold, an error linked to the current state will be generated. This situation indicates that an agent performed an unexpected behavior and the action was not successful executed. For example, if the multiagent system does not self-organize to a satisfactory solution, it will not produce the log “OBSERVER.observer.achieveEnergyTarget.#”. Thus, an error linked to the state “calculateEnergy” will be generated, as depicted in Figure 5.34.

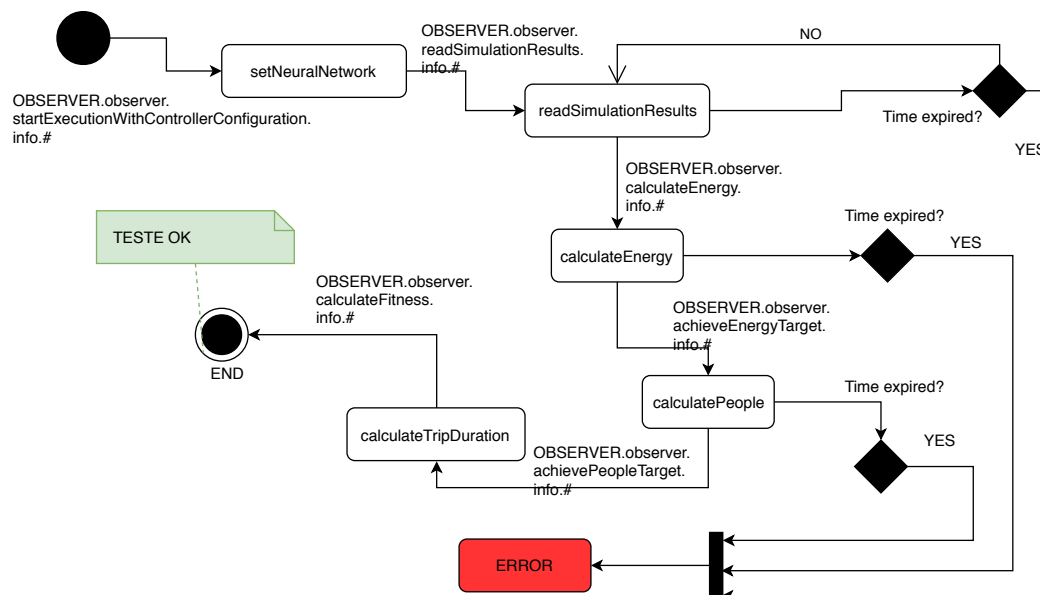


Figure 5.34: Simplified state machine for verifying test cases generated for the function “evaluate selected solution”.

Table 5.10: Functional tests at different perspectives (Simplified Table)

Perspective	Func.	Procedure	Input	Expected value	Validation Method (logs sorted into a timeline)
Designing and Framework	create Adaptive Agent to the streetlight	Manager Agent creates a new Adaptive Agent to the streetlight	Neural Network configuration (number of inputs and outputs)	Adaptive Agent with the selected neural network	1)MANAGER. receiveMsgFromSmartThing. *.*.*.smartThing.# 2)MANAGER. createAdaptiveAgent.INFO.# 3)AdaptiveAgent.lightsAgent. connect.# 4)MANAGER. sendMsgToSmartThing.INFO.# 5)AdaptiveAgent.lightsAgent. receiveInputData FromSmartThing.#
Behavior	collect data	streetlight 10 (node10) reads its sensors data	streetlight's motion and light sensors, and communication input	Adaptive Agent receives data from the streetlight's sensors	1)lightContainer.node10. receiveWirelessData.# 2)lightContainer.node10. readLightSensor.# 3)lightContainer.node10. readMotionSensor.# 4)lightContainer.node10. sendMsg.*.*. msgAdaptiveAgent 5)AdaptiveAgent.lightsAgent. receiveInputData FromSmartThing.#
Behavior	process output	AdaptiveAgent uses a neural network to process sensors data and generate output	streetlight's sensors data	Adaptive agent calculates two outputs (led and wireless data)	1)AdaptiveAgent.lightsAgent. useControllerToGetOutput.# 2)AdaptiveAgent.lightsAgent. sendOutputToSmartThing.#
Behavior and Environment	switch the light ON	Streetlight Agent (node 10) switches the light ON	neural network's light output value is positive	node10's light sensor detects a value equal or higher than its light brightness	1)lightContainer.node10. receiveNeural NetworkCommand.# 2)lightContainer.node10. switchLightON.# 3)lightContainer.node10. detectLight.# 4)lightContainer.lights. finishSimulation.#
Learning	change the neural network	ObserverAgent uses an individual's genes to set the ANN weights (see subsection)	an individual from the current generation	the ANN weights sequence is equal to the current individual	1)OBSERVER. chooseAdaptationMethod.# 2)OBSERVER. selectNeuralConfiguration.# 3)OBSERVER. useIndividualGenesToANN.# 4)OBSERVER. startExecutionWithController Configuration.#
Task Evaluation	evaluate the selected solution	Observer Agent analyzes the energy consumption and whether everyone finished their routes during the selected solution	the best individual of the last generation	energy consumption is less than 70% and everybody finished their routes	1)OBSERVER. startExecutionWith ControllerConfiguration.# 2)OBSERVER. readSimulationResults.# 3)OBSERVER. calculateEnergy.# 4)OBSERVER. achieveEnergyTarget.# 5)OBSERVER. achievePeopleTarget.# 6)OBSERVER. calculateFitness.#

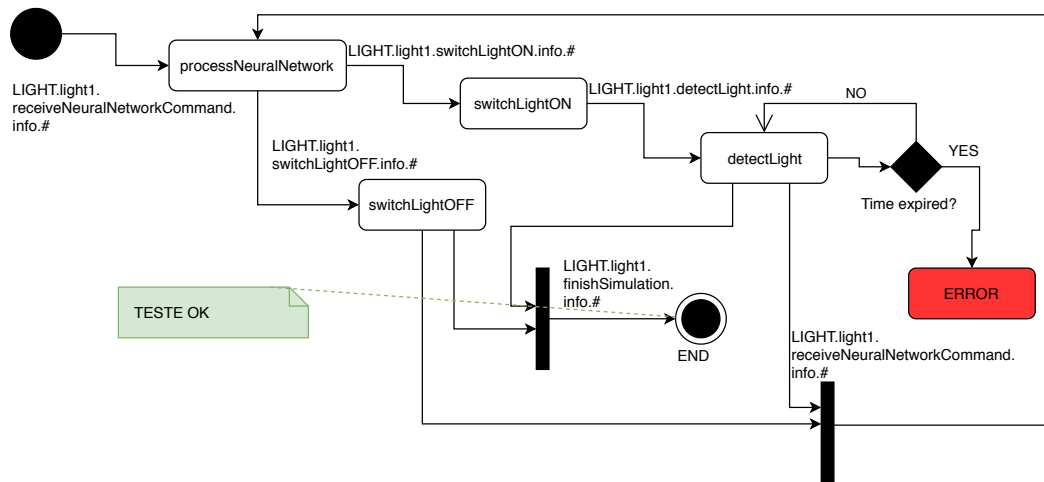


Figure 5.35: Simplified state machine for verifying test cases generated for the functions “switch the light ON” and “switch the light OFF”.

In order to force test failure and verify if these test applications were able to identify faults, we forced certain classes to act incorrectly during the execution of the program over some local tests. For example, to test the function “switch the light ON”, we inserted a defect that makes some streetlights to go dark during the simulation. Therefore, a streetlight agent that switched its light ON on the previous execution, did not detect brightness on the current execution and failed. As the test application did not receive the log “`LIGHT.light1.detectLight.info.#`”, its state machine indicated a failure in the state “`switchLightON`,” as depicted in Figure 5.36. Considering that a person can only move if his/her current and next positions are not completely dark, it interferes on the overall solution evaluation. Consequently, if a person does not finish his or her route, the test at the Task Evaluation perspective will also fail. Figure 5.37 depicts the logs that were generated by agents while this situation was being executed. Figure 5.38 depicts the global test that was executed without this defect.

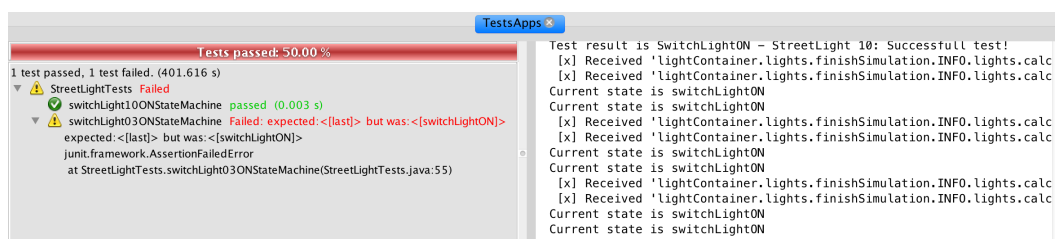


Figure 5.36: Executing the state machine to test the function “switch the light ON”: failure generated between states “`switchLightON`” and “`detectLight`” - specific log was not consumed.

Using our proposed solution, a test application can automatically select those logs from different agents that are essential for a specific test case and

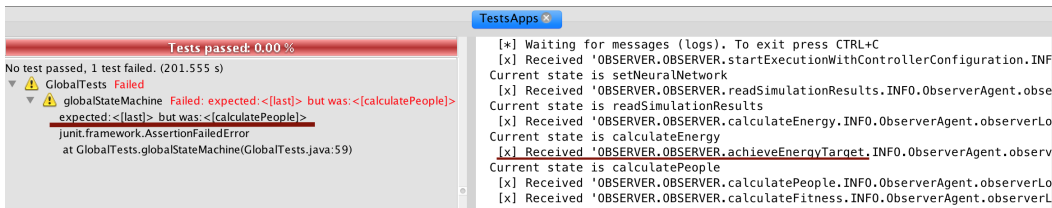


Figure 5.37: Executing the state machine to test the evaluation solution: failure generated between states “calculatePeople” and “calculateTripDuration” - because the machine did not receive the log that indicates that everyone finished their routes during the selected solution.

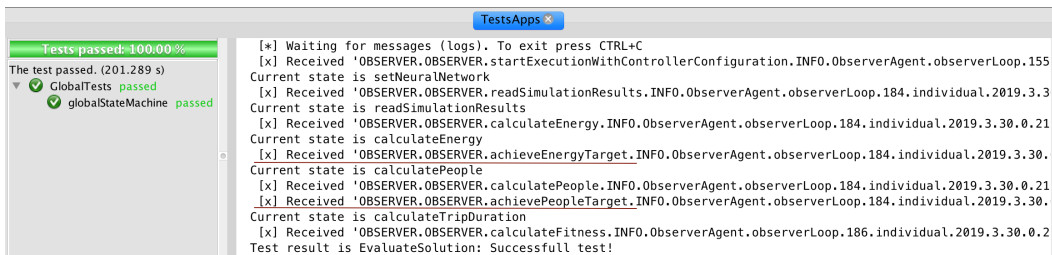


Figure 5.38: Executing the state machine to test the evaluation solution.

present them sorted in a single timeline. As a result, the interface depicted in Figure 5.39 shows just the logs that were consumed by the evaluation test application according to this binding key list. In addition, all logs are organized in a single timeline. As shown, not all logs depicted in Figure 5.40 were presented in this interface, but only the logs relevant to the execution of this test case. Thus, we were able to verify these logs in order to find the fault that generated the failure indicated by the state machine.

Agent Type	Class	Log Type	Action	Resource	Timestamp(ms)	Message
OBSERVER	ObserverAgent	INFO	startExecutionWithControllerConfiguration	individual	6.5090431E7	Selected individual: 1 2,1 6,-0 8,-0 8,1 6,1 5,-0 5,-0 3,-0 6,-0 9...
OBSERVER	ObserverAgent	INFO	calculateEnergy	individual	6.5264507E7	22 432659932659938
OBSERVER	ObserverAgent	INFO	achieveEnergyTarget	individual	6.5264592E7	target is 70%
OBSERVER	ObserverAgent	INFO	calculatePeople	individual	6.5264693E7	100 0
OBSERVER	ObserverAgent	INFO	achievePeopleTarget	individual	6.5264761E7	target is 100%
OBSERVER	ObserverAgent	INFO	calculateFitness	individual	6.5264842E7	47 693602693602685

Figure 5.39: Subscribing to receive only logs related to the evaluation solution testing.

Agent Type	Class	Log Type	Action	Resource	Timestamp(ms)	Message
MANAGER	ManagerAgent	INFO	connect	agent	6.04245E7	Agent started: (agent-identifier :name MANAGER@139 82 153 88:1...
OBSERVER	ObserverAgent	INFO	connect	agent	6.0424583E7	Agent started: (agent-identifier :name OBSERVER@139 82 153 88:1...
OBSERVER	ObserverAgent	INFO	chooseAdaptationMethod	agent	6.0424839E7	Method in use is Genetic Algorithm
OBSERVER	ObserverAgent	INFO	startExecutionWithControllerConfiguration	individual	6.0424923E7	Selected individual: 1 2,1 6,-0 8,-0 8,1 6,1 5,-0 5,-0 3,-0 6,-0 9...
OBSERVER	ObserverAgent	INFO	readSimulationResults	individual	6.0424978E7	
lightContainer	lights	INFO	calculateEdgeLight	edge	6.0474548E7	Light 0 0: Env Light 0 0 NBegin node7-0 0 NEnd node8-0 0
lightContainer	lights	INFO	connect	streetlight	6.0474548E7	
lightContainer	lights	INFO	calculateEdgeLight	edge	6.0474838E7	Light 0 0: Env Light 0 0 NBegin node8-0 0 NEnd node7-0 0
lightContainer	lights	INFO	calculateEdgeLight	edge	6.0474977E7	Light 0 0: Env Light 0 0 NBegin node7-0 0 NEnd node5-0 0
lightContainer	lights	INFO	calculateEdgeLight	edge	6.0475075E7	Light 0 0: Env Light 0 0 NBegin node5-0 0 NEnd node7-0 0
lightContainer	lights	INFO	calculateEdgeLight	edge	6.0475125E7	Light 0 0: Env Light 0 0 NBegin node7-0 0 NEnd node4-0 0
lightContainer	lights	INFO	calculateEdgeLight	edge	6.0475177E7	Light 0 0: Env Light 0 0 NBegin node4-0 0 NEnd node2-0 0
lightContainer	lights	INFO	calculateEdgeLight	edge	6.047527E7	Light 0 0: Env Light 0 0 NBegin node4-0 0 NEnd node7-0 0
lightContainer	lights	INFO	calculateEdgeLight	edge	6.0475297E7	Light 0 0: Env Light 0 0 NBegin node3-0 0 NEnd node2-0 0
lightContainer	lights	INFO	calculateEdgeLight	edge	6.0475396E7	Light 0 0: Env Light 0 0 NBegin node2-0 0 NEnd node4-0 0
lightContainer	lights	INFO	sendMsg	ManagerAgent	6.0475408E7	lightNeuralNetwork
lightContainer	lights	INFO	calculateEdgeLight	edge	6.0475488E7	Light 0 0: Env Light 0 0 NBegin node2-0 0 NEnd node1-0 0
MANAGER	ManagerAgent	INFO	receiveMsgFromSmartThing	smartThing	6.0475435E7	Received message from: lights
MANAGER	ManagerAgent	INFO	createAdaptiveAgent	adaptiveAgent	6.0475617E7	Create adaptive agent at: 169 254 249 202-lightsAgent with controll...

Figure 5.40: Subscribing to receive logs from all agents.

As shown in Table 5.10, we executed some functional tests at different perspectives. By using state machines, the test applications were able to validate these test cases by comparing the logs consumed from the MAS publisher against the logs listed in the “Validation Method” column. In addition, we also conducted some tests by inserting software failures and verifying if our test software could be useful for detecting these faults. As a result, after the state machine had indicated a failure, the developer could use the interface to identify the fault and reduce the diagnosis time.

5.3.4

How the proposed testing approach makes it possible to test an approach that adheres to the reference model

Table 5.11: Case III: Main statechart components.

Statechart Components		Approach
Agent Configuration	Body	Developing tests to the designing perspective, evaluating the sensors and actuators were selected to compose the agent.
	Controller	Developing tests to the designing perspective, evaluating the neural network that was selected to compose the agent.
Agent Behavior		Developing tests to the scenario and behavior perspectives, evaluating if agents are sensing and acting based on their body and controller characteristics, and in accordance with the environment changes.
Environment		Developing tests to the scenario perspective and framework perspectives, verifying communication among agents, and perturbatory channels that exist between agents and the environment.
Task Evaluation		Developing tests at the global level, investigating the performance of the group of agents in the application scenario. Basically, this test consumes logs from the ObserverAgent, which implements the Task Evaluation component and the learning algorithm.

Table 5.11 shows how this proposed testing approach makes it possible to test approaches that adhere to the proposed reference model. According to our

proposal, testing embodied agents requires to test each one of the statechart components. For example, to test the component “agent configuration”, we can develop tests to the designing perspective, evaluating the sensors and actuators that were selected to compose the agent.

6

Conclusion

Embodied agents have recently been proposed in several domains, such as health care, smart cities and agriculture. However, deploying these applications in specific scenarios has been very challenging because of the complex static and dynamic variability of the physical devices (e.g. sensors and actuators), the software scenario behavior and the environment.

According to existing experiments and our experience with embodied agents, we identified three main variation points to handle in order to create an embodied agent, including the variants that can be involved in a neural network design: (i) the body variability (i.e. number, types and brands of sensors and actuators); (ii) the complexity of the behavior of the agent, which varies based on the physical components that are operated by the agent (e.g. if this agent is able to communicate, the number of signals agents are able to exchange); and (iii) the agent controller that allows the agent to sense the environment and behave accordingly (e.g. if this agent controller is a neural network, in terms of its architectural variability the type of activation function and the number of neurons should be considered). In addition, the environment brings about variations in each one of these variation points, requiring the development of embodied agents that can reconfigure according to an evaluation criteria, such as performance and safety.

As formal methods, such as Statecharts, can endow agent-based systems with more intuitive and clear specifications, as well as specifications that are amenable to formal verification and analysis, favoring the development process of the whole system, we decided to investigate existing approaches for providing a reference model for self-configurable IoT embodied agents. Many modeling approaches to understand the relationship between the properties of agents and environments were proposed, as approaches to model reconfigurable systems. However, the use of reconfigurable systems is a challenge when specifying the behaviour of embodied agents. Therefore, we identified the main characteristics for both agent-based and reconfigurable systems, and used them as a basis to propose a reference model for reconfigurable embodied agents based on statecharts. Our focus was not on providing a highly elaborated

description, covering all possibly occurring concepts, but one that fits best a broad variety of embodied agent models in different domains, as required by the varied applications of Internet of Things (IoT). Although our reference model proposal is on high level of abstraction, we argue that it contributes particularly to a clear conceptualization of reconfigurable embodied agents, clarifying the relationship between the body and the control of an embodied agent and the complex and dynamic interactions between agents and their environments.

We have introduced this reference model so that it could serve to guide the development of software approaches that support the designing and testing of embodied agents. Based on the proposed reference model, we were able to develop three software approaches to embodied agents: (i) a software framework for the development of IoT embodied agents; (ii) an approach to configure embodied agents according to the environment; and (iii) an approach for testing reconfigurable embodied agents.

To validate each one of the software approaches, we developed some embodied agents applications to different scenarios. We have described some of these applications in this thesis work, since we provided other applications in (Nascimento et al., 2015; Nascimento Marx Leles Viana, 2016; Nascimento and Lucena, 2017; Nascimento and Lucena, 2017).

We used the application scenario of Smart Street Lights to illustrate different aspects of this thesis work. First, we introduced this case study for showing an application of the proposed reference model. Second, to illustrate the use of the software framework, we instantiated this application. Third, to illustrate the use of the approach to configure embodied agents according to the environment, we reproduced this street light application scenario, but showing how this application operates in a dynamic environment. Finally, we showed how our proposed testing approach can be used for testing agents embodied in smart street lights scenarios.

We believe that our approach can further assist a software engineer in achieving a broader understanding of embodied agents configurations and their effect under different environments.

7

Future Work and Open Challenges

Many possible future directions stem from this work. A first interesting direction is to extend the current description to address other aspects of MAS development, such as agents' communication and other kinds of interactions between agents. In addition, as we provided a reference model that is on high level of abstraction, it can also be extended in order to fit best a specific application domain, such as smart cities.

In addition, the applicability of the conceptual model presented in this paper is not strictly limited to our implementation approaches and could be reused in further research, as a foundation for the development of new approaches for embodied agents.

In the following sections, we describe other interesting directions for the development of reconfigurable embodied agents.

7.1

Morphology-based agent design

(Auerbach and Bongard, 2009) discuss the need of approaches to design embodied agents solutions that are able to evolve morphologically, that is, adapting the evolutionary algorithm to also include the possible agent body configurations. In addition, (Auerbach and Bongard, 2009) also argue that there is no standard about how the controller should be organized and about whether it is necessary to use a structural modularity in controller design.

7.1.1

Other learning algorithms

We designed our approaches taking into consideration the main adaptive approaches for the controller of embodied agents, that are the evolutionary and back-propagation algorithms. We can conduct more experiments using other learning algorithms in order to construct a sufficient and representative model. Note that our model is generic and we may consider other alternative prediction modules and learning algorithms.

7.2

Descriptive Evaluations

(Auerbach and Bongard, 2012) performed an experiment to investigate interactions between body and environment, but they state that this relationship is not well understood. According to the authors, there is a need of investigating how the body of evolving robots varies in other environments, and investigating which environments drive an increase in body complexity and which ones increase the complexity of the control strategy. They also suggest to investigate if it is possible to measure the complexity of robots while they are evolving in order to affect the evolutionary search process.

Therefore, this is clearly a future work that we will address. To measure the complexity of embodied agents while they are being adapted, we will maintain a profile of environmental changes and body and controller reconfigurations. Therefore, we can investigate which types of reconfiguration are usually triggered by specific environment changes. The metadata-driven and publish-subscribe-based approach that we implemented to test and evaluate embodied agents will assist us with these descriptive evaluations.

7.3

Embodied agent testing and verification

According to (Bredeche et al., 2018), there is a gap of methods and tools for analyzing the evolutionary dynamics at work, which is a technique usually used to evolve embodied agents. As proposed in the previous section, we can also use our publish-subscribe-based approach for analyzing the evolutionary dynamics at work, bearing on how the robots behave and change their behavior when deployed.

In addition, based on our proposed statecharts, we also aim at delivering a solution for the formal verification of embodied agents.

8

Bibliography

AGRE, P. E. Computational research on interaction and agency. **Artificial intelligence**, Elsevier, v. 72, n. 1-2, p. 1–52, 1995.

ARAÚJO, T. P. de; STAA, A. von. Supporting failure diagnosis with logs containing meta-information annotations. **Technical Reports in Computer Science. PUC-Rio. ISSN 0103-9741**, PUC-Rio, v. 14, p. 21, 2014.

ARDUINO. **Arduino**. [Http://www.arduino.cc/](http://www.arduino.cc/).

ATZORI, L. et al. The social internet of things (siot)–when social networks meet the internet of things: Concept, architecture and network characterization. **Computer networks**, Elsevier, v. 56, n. 16, p. 3594–3608, 2012.

AUERBACH, J.; BONGARD, J. C. Evolution of functional specialization in a morphologically homogeneous robot. In: **Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation**. New York, NY, USA: ACM, 2009. (GECCO '09), p. 89–96. ISBN 978-1-60558-325-9. Disponível em: <<http://doi.acm.org/10.1145/1569901.1569915>>.

AUERBACH, J. E.; BONGARD, J. C. On the relationship between environmental and morphological complexity in evolved robots. In: **Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation**. New York, NY, USA: ACM, 2012. (GECCO '12), p. 521–528. ISBN 978-1-4503-1177-9. Disponível em: <<http://doi.acm.org/10.1145/2330163.2330238>>.

AYALA, I. et al. A software product line process to develop agents for the iot. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 15, n. 7, p. 15640–15660, 2015.

BAE, J. W.; MOON, I.-C. Ldef formalism for agent-based model development. **IEEE Transactions on Systems, Man, and Cybernetics: Systems**, IEEE, v. 46, n. 6, p. 793–808, 2015.

BANARSE, D. et al. The body is not a given: Joint agent policy learning and morphology evolution. In: INTERNATIONAL FOUNDATION FOR AUTONOMOUS

AGENTS AND MULTIAGENT SYSTEMS. **Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems**. [S.l.], 2019. p. 1134–1142.

BARESI, L.; GUINEA, S.; SHAHZADA, A. Short paper: Harmonizing heterogeneous components in sesame. In: IEEE. **Internet of Things (WF-IoT), 2014 IEEE World Forum on**. [S.l.], 2014. p. 197–198.

BEER, R. D. The dynamics of brain–body–environment systems: A status report. In: **Handbook of Cognitive Science**. [S.l.]: Elsevier, 2008. p. 99–120.

BELEW, R. K.; MCINERNEY, J.; SCHRAUDOLPH, N. N. Evolving networks: Using the genetic algorithm with connectionist learning. In: CITESEER. In. [S.l.], 1990.

BELLIFEMINE, F. et al. **Jade Administrator's Guide**. jade.tilab.com/doc/administratorsguide.pdf, 2007.

BELLIFEMINE, F. et al. **Jade Programmer's Guide**. jade.tilab.com/doc/programmersguide.pdf, April 2010.

BOE, A.; SALUNKHE, D. Ripening tomatoes: Ethylene, oxygen, and light treatments. **Economic Botany**, Springer, v. 21, n. 4, p. 312–319, 1967.

BREDECHE, N.; HAASDIJK, E.; PRIETO, A. Embodied evolution in collective robotics: a review. **Frontiers in Robotics and AI**, Frontiers, v. 5, p. 12, 2018.

BRIOT, J.-P.; NASCIMENTO, N. M. de; LUCENA, C. J. P. de. A multi-agent architecture for quantified fruits: Design and experience. In: SEKE/KNOWLEDGE SYSTEMS INSTITUTE, PA, USA. **28th International Conference on Software Engineering & Knowledge Engineering (SEKE'2016)**. [S.l.], 2016. p. 369–374.

BROOKS, R. A. Intelligence without reason. **The artificial life route to artificial intelligence: Building , situated agents**, Lawrence Erlbaum Associates Hillsdale, New Jersey, p. 25–81, 1995.

BRUNI, R. et al. A white box perspective on behavioural adaptation. In: **Software, Services, and Systems**. [S.l.]: Springer, 2015. p. 552–581.

CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. **ACM Computing Surveys (CSUR)**, ACM, v. 28, n. 4, p. 626–643, 1996.

D'INVERNO, M.; LUCK, M.; LUCK, M. M. **Understanding agent systems**. [S.l.]: Springer Science & Business Media, 2004.

FIPA. **The Foundation for Intelligent Physical Agents**. 08 2015. [Http://www.fipa.org/](http://www.fipa.org/).

FLOREANO, D.; MATTIUSI, C. **Bio-Inspired Artificial Intelligence. Theories, Methods, and Technologies**. [S.l.]: Cambridge: MIT Press, 2008.

FRANKLIN, S. Autonomous agents as embodied ai. **Cybernetics & Systems**, Taylor & Francis, v. 28, n. 6, p. 499–520, 1997.

FUKAI, T.; TANAKA, S. A simple neural network exhibiting selective activation of neuronal ensembles: from winner-take-all to winners-share-all. **Neural computation**, MIT Press, v. 9, n. 1, p. 77–97, 1997.

GALSTER, M. et al. Variability in software systems—a systematic literature review. **IEEE Transactions on Software Engineering**, IEEE, v. 40, n. 3, p. 282–306, 2014.

GOOGLE. Google trends. <https://trends.google.com/>, January 2018.

HAREL, D. Statecharts: A visual formalism for complex systems. **Science of computer programming**, Elsevier, v. 8, n. 3, p. 231–274, 1987.

HAREL, D. et al. Labor division with movable walls: Composing executable specifications with machine learning and search. 2019.

HAYKIN, S. **Neural Networks: A Comprehensive Foundation**. [S.l.]: Macmillan, 1994. ISBN 9780023527616.

HERRERO-PEREZ, D.; MARTINEZ-BARBERA, H. Decentralized coordination of automated guided vehicles (short paper). In: **AAMAS 2008**. [S.l.: s.n.], 2008.

HORN, P. Autonomic computing: Ibm\'s perspective on the state of information technology. IBM, 2001.

INGRAND, F. Recent trends in formal validation and verification of autonomous robots software. In: **IEEE International Conference on Robotic Computing**. [S.l.: s.n.], 2019.

JARRAYA, A. et al. Distributed collaborative reasoning for har in smart homes (extended abstract). In: IFAAMAS. **Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)**. [S.l.], 2018. p. 1971–1973.

JELISAVCIC, M.; ROIJERS, D. M.; EIBEN, A. Analysing the relative importance of robot brains and bodies. In: MIT PRESS. **Artificial Life Conference Proceedings**. [S.l.], 2018. p. 327–334.

JOHNSON, D.; HIPPS, N.; HAILS, S. **Helping Consumers Reduce Fruit and Vegetable Waste: Final Report**. [S.l.], 2008.

KARSAI, G.; SZTIPANOVITS, J. A model-based approach to self-adaptive software. **IEEE Intelligent Systems and Their Applications**, IEEE, v. 14, n. 3, p. 46–53, 1999.

KATASONOV, A. et al. Smart semantic middleware for the internet of things. **ICINCO-ICSO**, v. 8, p. 169–178, 2008.

KINNY, D. Reliable agent computation: An algebraic approach. In: SPRINGER. **Pacific Rim International Workshop on Multi-Agents**. [S.l.], 2001. p. 31–47.

KLÜGL, F.; DAVIDSSON, P. Amason: Abstract meta-model for agent-based simulation. In: SPRINGER. **German Conference on Multiagent System Technologies**. [S.l.], 2013. p. 101–114.

KUURKOVA, V. Kolmogorov's theorem and multilayer neural networks. **Neural networks**, Elsevier, v. 5, n. 3, p. 501–506, 1992.

LAPOUCHNIAN, A. et al. Requirements-driven design of autonomic application software. In: IBM CORP. **Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research**. [S.l.], 2006. p. 7.

LUCENA, C. **Software engineering for multi-agent systems II: research issues and practical applications**. [S.l.]: Springer Science & Business Media, 2004.

LUCK, M.; D'INVERNO, M. et al. A formal framework for agency and autonomy. In: **ICMAS**. [S.l.: s.n.], 1995. v. 95, p. 254–260.

LUCKCUCK, M. et al. Formal specification and verification of autonomous robotic systems: A survey. **arXiv preprint arXiv:1807.00048**, 2018.

MACDONALD, B.; HSIEH, B.; WARREN, I. Design for dynamic reconfiguration of robot software. In: **Proceedings of the Second International Conference on Autonomous Robots and Agents (ICARA 2004)**. [S.l.: s.n.], 2004.

MACÍAS-ESCRIVÁ, F. D. et al. Self-adaptive systems: A survey of current approaches, research challenges and applications. **Expert Systems with Applications**, Elsevier, v. 40, n. 18, p. 7267–7279, 2013.

MAROCCO, D.; NOLFI, S. Emergence of communication in embodied agents evolved for the ability to solve a collective navigation problem. **Connection Science**, Taylor & Francis, v. 19, n. 1, p. 53–74, 2007.

MAROCCO, D.; NOLFI, S. Emergence of communication in embodied agents evolved for the ability to solve a collective navigation problem. **Connection Science**, 2007.

MARZO, G. D. et al. **Engineering Self-Organising Systems**. Berlin: Springer, 2004.

MASSERA, G. et al. Farsa: An open software tool for embodied cognitive science. In: **Advances in Artificial Life, ECAL**. [S.l.: s.n.], 2013. v. 12, p. 538–545.

MASSERA, G. et al. Designing adaptive humanoid robots through the farsa open-source framework. **Adaptive Behavior**, SAGE Publications Sage UK: London, England, v. 22, n. 4, p. 255–265, 2014.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, Springer, v. 5, n. 4, p. 115–133, 1943.

MILLER, G. F.; TODD, P. M.; HEGDE, S. U. Designing neural networks using genetic algorithms. In: MORGAN KAUFMANN PUBLISHERS INC. **Proceedings of the third international conference on Genetic algorithms**. [S.l.], 1989. p. 379–384.

NASCIMENTO MARX LELES VIANA, C. J. P. d. L. Nathalia Moraes do. An iot-based tool for human gas monitoring. In: CBIS 2016 (ISSN 2178-2857). **Congresso Brasileiro de Informática em Saúde - CBIS 2016**. [S.l.], 2016. v. 15, p. 96–98.

NASCIMENTO, N. A self-configurable iot agent system based on environmental variability. In: INTERNATIONAL FOUNDATION FOR AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems**. [S.l.], 2018. p. 1761–1763.

NASCIMENTO, N. et al. A context-aware machine learning-based approach. In: ACM. **Computer Science and Software Engineering (CASCON), 28th Annual International Conference on**. [S.l.], 2018.

NASCIMENTO, N. et al. A context-aware machine learning-based approach. In: IBM CORP. **Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering**. [S.l.], 2018. p. 40–47.

Nascimento, N. et al. Machine Learning-based Variability Handling in IoT Agents. **ArXiv e-prints**, fev. 2018.

NASCIMENTO, N. et al. Testing self-organizing multiagent systems. **arXiv preprint arXiv:1904.01736**, 2019.

NASCIMENTO, N. et al. A publish-subscribe based architecture for testing multi-agent systems. In: SEKE/KNOWLEDGE SYSTEMS INSTITUTE, PA, USA. **29th International Conference on Software Engineering & Knowledge Engineering (SEKE'2017)**. [S.l.], 2017.

NASCIMENTO, N. M. do; LUCENA, C. J. P. de. Engineering cooperative smart things based on embodied cognition. In: IEEE. **Adaptive Hardware and Systems (AHS), 2017 NASA/ESA Conference on**. [S.l.], 2017. p. 109–116.

NASCIMENTO, N. M. do; LUCENA, C. J. P. de. Fiot: An agent-based framework for self-adaptive and self-organizing applications based on the internet of things. **Information Sciences**, Elsevier, v. 378, p. 161–176, 2017.

NASCIMENTO, N. M. do; LUCENA, C. J. P. de; FUKS, H. Modeling quantified things using a multi-agent system. In: IEEE. **Web Intelligence and Intelligent Agent Technology (WI-IAT), 2015 IEEE/WIC/ACM International Conference on**. [S.l.], 2015. v. 1, p. 26–32.

NELSON, A.; BARLOW, G.; DOITSIDIS, L. Fitness functions in evolutionary robotics: A survey and analysis. **Robotics and Autonomous Systems**, 2007.

NOLFI, S. **Laboratory of Autonomous Robotics and Artificial Life**. <http://laral.istc.cnr.it/>, March 1995.

NOLFI, S. et al. Evolutionary robotics. In: _____. **Springer Handbook of Robotics**. Cham: Springer International Publishing, 2016. cap. 76, p. 2035–2068. ISBN 978-3-319-32552-1.

NOLFI, S.; FLOREANO, D. Coevolving predator and prey robots: Do “arms races” arise in artificial evolution? **Artificial life**, MIT Press, v. 4, n. 4, p. 311–335, 1998.

NOLFI, S.; FLOREANO, D. **Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines**. Cambridge, MA, USA: MIT Press, 2000. ISBN 0262140705.

NOLFI, S.; PARISI, D. Learning to adapt to changing environments in evolving neural networks. In: **Adaptive Behavior**. [S.l.: s.n.], 1997. p. 75–98.

OLIVEIRA, E.; LOULA, A. Symbol interpretation in neural networks: an investigation on representations in communication. In: **Proceedings of the Annual Meeting of the Cognitive Science Society**. [S.l.: s.n.], 2014. v. 36, n. 36.

OLIVEIRA, E. S. de; LOULA, A. Symbolic associations in neural network activations: Representations in the emergence of communication. In: IEEE. **Neural Networks (IJCNN), 2015 International Joint Conference on**. [S.l.], 2015. p. 1–8.

PAGLIUCA, P.; MILANO, N.; NOLFI, S. Maximizing adaptive power in neuroevolution. **PloS one**, Public Library of Science, v. 13, n. 7, p. e0198788, 2018.

PĚCHOUČEK, M.; MAŘÍK, V. Industrial deployment of multi-agent technologies: review and selected case studies. **Autonomous Agents and Multi-Agent Systems**, Springer, v. 17, n. 3, p. 397–431, 2008.

PEZZULO, G.; NOLFI, S. Making the environment an informative place: A conceptual analysis of epistemic policies and sensorimotor coordination. **Entropy**, Multidisciplinary Digital Publishing Institute, v. 21, n. 4, p. 350, 2019.

POHL, K.; BÖCKLE, G.; LINDEN, F. J. van D. **Software product line engineering: foundations, principles and techniques**. [S.l.]: Springer Science & Business Media, 2005.

POLANI, D. An informational perspective on how the embodiment can relieve cognitive burden. In: IEEE. **Artificial Life (ALIFE), 2011 IEEE Symposium on**. [S.l.], 2011. p. 78–85.

POSLAD, S. Specifying protocols for multi-agent systems interaction. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM, v. 2, n. 4, p. 15, 2007.

POSTCAPES. **Tracking the Internet of Things**. August 2019. [Http://postscapes.com/categories](http://postscapes.com/categories).

QUICK, T. et al. On bots and bacteria: Ontology independent embodiment. In: SPRINGER. **European Conference on Artificial Life**. [S.l.], 1999. p. 339–343.

RABBITMQ. **RabbitMQ**. 10 2016. Available in <https://www.rabbitmq.com/>.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **Nature**, v. 323, n. 6088, p. 533–536, 1986.

RUSSELL, S.; NORVIG, P. Artificial intelligence: a modern approach. 1995.

SANTOS, F.; NUNES, I.; BAZZAN, A. L. Model-driven engineering in agent-based modeling and simulation: a case study in the traffic signal control domain. In: INTERNATIONAL FOUNDATION FOR AUTONOMOUS AGENTS AND MULTI-AGENT SYSTEMS. **Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems**. [S.l.], 2017. p. 1725–1727.

SESHIA, S. A.; SADIGH, D.; SASTRY, S. S. Towards verified artificial intelligence. **arXiv preprint arXiv:1606.08514**, 2016.

SOBE, A.; FEHERVARI, I.; ELMENREICH, W. Frevo: A tool for evolving and evaluating self-organizing systems. In: **IEEE Self-adaptive and Self-organizing Systems Workshop**. [S.l.: s.n.], 2012.

SOMMERVILLE, I. **Software Engineering**. Pearson/Addison-Wesley, 2004. (International computer science series). ISBN 9780321210265. Disponível em: <<http://books.google.com.br/books?id=fIJQAAAAMAAJ>>.

SONI, G.; KANDASAMY, S. Smart garbage bin systems—a comprehensive survey. In: SPRINGER. **International Conference on Intelligent Information Technologies**. [S.l.], 2017. p. 194–206.

SPIVEY, J. M. **Understanding Z: a specification language and its formal semantics**. [S.l.]: Cambridge University Press, 1988.

STARUML. **Star UML Tool**. August 2019. [Http://staruml.io/](http://staruml.io/).

STEELS, L. **ECAGENTS: Embodied and Communicating Agents**. [S.l.], 2004.

SWAN, M. Sensor mania! The Internet of Things, wearable computing, objective metrics, and the Quantified Self 2.0. **Journal of Sensor and Actuator Networks**, v. 1, n. 3, p. 217–253, 2012.

SWAN, M. Connected car: Quantified self becomes quantified car. **Journal of Sensor and Actuator Networks**, Multidisciplinary Digital Publishing Institute, v. 4, n. 1, p. 2–29, 2015.

TELECOM. **JAVA Agent DEvelopment Framework**. 08 2015. [Http://jade.tilab.com/](http://jade.tilab.com/).

THRUN, S. et al. Stanley: The robot that won the darpa grand challenge. In: **The 2005 DARPA Grand Challenge**. [S.l.]: Springer, 2007. p. 1–43.

TURNER, A. J.; MILLER, J. F. Neuroevolution: evolving heterogeneous artificial neural networks. **Evolutionary Intelligence**, Springer, v. 7, n. 3, p. 135–154, 2014.

VIDE, M. D.; NOLFI, S. Emergence of communication in teams of embodied and situated agents. In: WORLD SCIENTIFIC. **The Evolution of Language: Proceedings of the 6th International Conference (EVLANG6), Rome, Italy, 12-15 April 2006**. [S.I.], 2006. p. 198.

WEYNS, D.; MALEK, S.; ANDERSSON, J. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM, v. 7, n. 1, p. 8, 2012.

WHITE, J. E. Telescript technology: The foundation for the electronic marketplace. **General Magic white paper**, v. 282, 1994.

WHITESON, S. et al. Evolving soccer keepaway players through task decomposition. **Machine Learning**, Springer, v. 59, n. 1-2, p. 5–30, 2005.

WOLPER, P. Temporal logic can be more expressive. **Information and control**, Elsevier, v. 56, n. 1-2, p. 72–99, 1983.

WOOLDRIDGE, M. **An introduction to multiagent systems**. [S.I.]: John Wiley & Sons, 2009.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: Theory and practice. **The knowledge engineering review**, Cambridge Univ Press, v. 10, n. 02, p. 115–152, 1995.

YAO, X. Evolving artificial neural networks. **Proceedings of the IEEE**, IEEE, v. 87, n. 9, p. 1423–1447, 1999.

YIM, M. et al. Modular self-reconfigurable robot systems [grand challenges of robotics]. **IEEE Robotics & Automation Magazine**, IEEE, v. 14, n. 1, p. 43–52, 2007.

ZEDADRA, O. et al. Towards a reference architecture for swarm intelligence-based internet of things. In: SPRINGER. **International Conference on Internet and Distributed Computing Systems**. [S.I.], 2017. p. 75–86.

ZHU, L.; CAI, H.; JIANG, L. Minson: A business process self-adaptive framework for smart office based on multi-agent. In: IEEE. **e-Business Engineering (ICEBE), 2014 IEEE 11th International Conference on**. [S.I.], 2014. p. 31–37.