School of Technology and Architecture

Learning Control Knowledge by Observation in Software Agents

Paulo Roberto Almeida Moreira Costa

A Thesis presented in partial fulfilment of the Requirements for the Degree of

Doctor in Information Science and Technology

Jury:

Doutora Ana Maria Severino de Almeida e Paiva, Prof. Associada, IST-UL
Doutor Francisco Saraiva de Melo, Prof. Auxiliar, IST-UL
Doutor Rui Carlos Camacho de Sousa Ferreira da Silva, Prof. Associado, FEUP
Doutor Anders Lyhne Christensen, Prof. Auxiliar, ISCTE-IUL

July, 2013

”We can only see a short distance ahead, but we can see plenty there that needs to be done.”

Alan Turing

Acknowledgements

I would like to take this opportunity to thank everyone who played an important role in the completion of this thesis.

First of all, I would like to express my sincerest gratitude to my supervisor, Professor Luís Botelho for his generous guidance and enthusiasm throughout this academic journey.

I would also like to thank my family and my friends for all the support, especially for never letting me down and always believing in me.

I am also very grateful to ADETTI-IUL and IT for grating me this project and for all the support, and to FCT for the scholarship they have provided, which supported my expenses throughout the entire project and allowed me to completely dedicate myself to it.

Finally, I would like to give a special thanks to Adriano for his support and companionship throughout the final correction of my thesis.

To anyone else I might not have mentioned, who helped me directly or indirectly, please know that I am very thankful.

This thesis reports PhD research work, for the Doctoral Program on Information Science and Technology of ISCTE-Instituto Universitario de Lisboa. It is partially supported by Fundação para a Ciencia e a Tecnologia through the PhD Grant number SFRH/BD/44779/2008 and the Associated Laboratory number 12 - Instituto de Telecomunicações - PEst-OE/EEI/LA0008/2013.

Abstract

This thesis is the outcome of research on providing software agents with learning by observation capabilities. It presents an agent architecture that allows software agents to learn control knowledge by direct observation of the actions executed by expert agents while performing a task. The proposed architecture makes it possible for software agents to observe each other. It displays information that is essential for observation, such as the agent constituents and capabilities, the actions performed and the conditions holding for them. The displayed information is accessible to all agents that want to observe.

The proposed approach combines two methods of learning from the observed data. The first one relies on the sequence which the actions were observed. The second one categorizes the information in the observed data and determines which set of categories the new problems belong. The two learning methods are incorporated into a learning process that covers all aspects of learning by observation such as the discovery and observation of experts, storage of the acquired information, learning and application of the acquired knowledge. The learning process also includes an evaluation of the agent's progress which provides control over the decision to obtain new knowledge or apply the acquired knowledge to new problems. The process is extended with external feedback on the actions executed by the agent.

The approach was tested on three different scenarios that show that learning by observation can be of key importance whenever agents sharing similar features want to learn from each other.

Keywords: learning by observation, software agents, machine learning, software image

Resumo

Esta tese resulta da investigação da aplicação da aprendizagem por observação em agentes de software. A tese apresenta uma arquitetura que permite a agentes de software aprender mecanismos de controlo por observação direta das ações realizadas por agentes especialistas enquanto estes realizam uma tarefa. A arquitetura proposta permite que agentes de software se observem uns aos outros ao exibir informações que são essenciais para a observação, tais como os constituintes e as capacidades do agente, as ações realizadas e as condições existentes aquando a realização das mesmas. Esta informação é acessvel a todos os agentes que queiram observar.

A abordagem proposta combina dois métodos de aprendizagem. O primeiro baseia-se na sequência em que as ações foram observadas. O segundo categoriza a informação observada e determina o conjunto de categorias aos quais os novos problemas pertencem. Os dois métodos de aprendizagem são incorporados num processo de aprendizagem que cobre todos os aspetos da aprendizagem por observação tais como a descoberta e observação de especialistas, o armazenamento da informação adquirida, a aprendizagem e a aplicação do conhecimento adquirido. O processo de aprendizagem inclui também uma avaliação do progresso do agente que controla a decisão de obter novo conhecimento ou de aplicar o conhecimento adquirido em novos problemas. O processo é alargado com *feedback* externo sobre as ações executadas.

A abordagem foi testada em três diferentes cenários que mostram a importância da aprendizagem por observação em situações onde agentes que compartilham características semelhantes querem aprender uns com os outros.

Palavras-chave: aprendizagem por observação, agentes de software, aprendizagem automática, imagem de software

Contents

Acknowledgements iii

Abstract v

Resumo vi

List of Tables x

List of Figures xii

1 Introduction 1

1.1 Motivation . . . 3

1.2 Approach Definition . . . 4

1.2.1 The Visible Representation of the Software Agent . . . 5

1.2.2 Learning by Observation of the Software Image . . . 9

1.2.3 Approach Contributions and Limitations . . . 12

1.3 Evaluation . . . 14

1.4 Demonstration Scenarios and Results . . . 15

2 State of the Art 19

2.1 The Software Visual Image . . . 22

2.1.1 Embodiment . . . 23

2.1.2 Code Analysis and Function Tracking . . . 24

2.1.3 Requirements on the Observed Data . . . 25

2.1.4 Representing Agent Actions . . . 26

2.1.5 The Point of View of the Observed Expert . . . 27

2.1.6 Previous Proposals on the Software Image . . . 28

2.2 Learning by Observation 29
2.2.1 Social Learning Theories 32
2.2.2 Possibilities for the Leaning Algorithm 33
2.2.3 Evaluating the Agent's Knowledge 40
2.2.4 The Mirror Neurons 41
2.2.5 Approaches to Learning by Observation 44
2.3 Critical Comments and Conclusions 45

3 The Software Image **49**
3.1 The Importance of the Agent Sensors and Perception 52
3.2 The Elements of the Software Image 55
3.2.1 The Static Image 56
3.2.2 The Dynamic Image 58
3.3 The Software Image Meta-ontology 60
3.4 The Software for the Software Image 63
3.4.1 Software Image Creation 66
3.4.2 Updating the Dynamic Image 70
3.4.3 Discovering and Observing the Software Image 73

4 Learning by Observation **75**
4.1 The Mirror Properties of the Learning Process 78
4.2 Architectural Components of the Learning Process 79
4.3 Discovering and Observing Software Agents 80
4.3.1 Discovering and Selecting Experts to Observe 82
4.3.2 Observing the Selected Expert 84
4.4 Storing the Information Acquired in Observation 85
4.4.1 The Internal Knowledge Representation 85
4.4.2 Storing Experiences in the Agent's Memory 88
4.5 Learning from the Information in the Agent's Memory 90
4.5.1 The Recall Method 93
4.5.2 The Classification Method 96
4.6 Evaluation of the Agent's Progress 97

5 Application Scenarios **102**

5.1 Tests on the Software Image 104

5.1.1 Creating and Observing the Software Image 105

5.1.2 Comparing Software Images 107

5.2 The Virtual Hand Scenario 111

5.2.1 Results for the First Two Settings 114

5.2.2 Results for the Last Two Settings 118

5.2.3 Setting the Configurable Parameters 121

5.2.4 The Influence of the External Feedback 124

5.2.5 The Impact of the Software Image Meta-ontology 126

5.3 The Calculator Scenario 129

5.4 The Mountain Car Scenario 137

5.5 Validation of the Evaluation Criteria 142

6 Conclusions and Future Work **144**

References **150**

A Embodiment and Embodied Cognition **160**

A.1 Embodiment Through History 160

A.2 The Grounding Problem 162

A.3 Computational Approaches for Embodiment 163

B Description of The Software Image Ontology **166**

C Testing Machine Learning Algorithms **169**

D Scenario Settings **174**

List of Tables

5.1 Description of the agents used in the test 105

5.2 Results for the test on creating the software image 106

5.3 Results for the test on discovering experts and observing the software image . . 106

5.4 The settings for the virtual hand scenario 113

5.5 The average weights of the recall and classification methods on each setting . . 114

5.6 Results of experimenting the first two settings of the virtual hand scenario . . . 115

5.7 The average weights of the recall and classification methods on each setting . . 118

5.8 Results of experimenting the last two settings of the virtual hand scenario . . . 119

5.9 Results of experimenting the fifth setting of the virtual hand scenario 124

5.10 Comparison between learning from *expert1* and from *expert2* 127

5.11 Comparison between learning from a task and from an identical agent 129

5.12 Overview of the results of the experiment of the number calculator scenario when facing the same conditions 132

5.13 Overview of the results of experimenting the number calculator scenario when facing different conditions 133

5.14 The time spent by a simulation step for the calculator scenario 136

5.15 Overview of the results of experimenting the mountain car scenario 139

C.1 The settings for the test scenarios. 170

C.2 The average time taken to learn from the examples and propose actions 173

List of Figures

3.1 The software image in relation with the software agent 50
3.2 How the activities of other agents can hinder the ability to determine the conditions 54
3.3 UML diagram of the software image 56
3.4 The object diagram of an example static image 58
3.5 The object diagram of an example of a snapshot 59
3.6 The software image meta-ontology 61
3.7 An example of use of the *datatype* property 62
3.8 An example of use of the *equivalentTo* relationship 62
3.9 The functionalities provided by the software for the software image 64
3.10 The annotations and their relation with the software image 65
3.11 An example of using the *argumentTypes* attribute for generic invocations 66
3.12 Creating snapshots and updating the dynamic image 72

4.1 The activities of the learning process on each state 76
4.2 The influence of the confidence thresholds in state transition 77
4.3 The components of the agent architecture 79
4.4 The process of discovering and observing experts 81
4.5 The concept of an observable expert 83
4.6 The EXPERIENCE and its relation with the snapshot 85
4.7 A representation of the structure holding the agent's memory 86
4.8 An example of the agent's memory 87
4.9 The action possibilities to be proposed by the recall method 95

5.1 The static images of agent A and $A*$ 107
5.2 The static images of agent B and C 108
5.3 The static images of agent D and E 108

5.4 Results for the first software image comparison test 109
5.5 Results for the second software image comparison test 110
5.6 The data-type of the *hand* visible attribute 112
5.7 The progress of the agent in *exp1* and *exp2* throughout the experiment 117
5.8 The progress of the agent in *exp3* and *exp4* throughout the experiment 120
5.9 The impact of changing the confidence thresholds 122
5.10 The progress of the agent in *exp5* throughout the experiment 125
5.11 The parts shared by the agent and *expert1* 127
5.12 Representation of the different designations for the visible attribute 128
5.13 Progress of the apprentice agents for the first setting of the calculator scenario . 133
5.14 Progress of the apprentice agents for the second setting of the calculator scenario 135
5.15 Results of experimenting the mountain car scenario - steps to reach the top . . . 140
5.16 Results of experimenting the mountain car scenario - distance travelled and number of times reached the top 141

B.1 The OWL specification of the software image meta-ontology 167
B.2 An example of an agent ontology in OWL 168

C.1 The results for first and second settings 171
C.2 The results for third and fourth settings 172

D.1 The component diagram of the elements of the virtual hand scenario 174
D.2 The class diagram of the two kinds of expert agents in the virtual hand scenario 175
D.3 The class diagram of the apprentice agent in the virtual hand scenario 175
D.4 The component diagram of the elements of the number calculator scenario . . . 176
D.5 The class diagram of the expert and the apprentice agent in the number calculator scenario 177
D.6 The component diagram of the elements of the mountain car scenario 178
D.7 The class diagram of the expert and the apprentice agents in the mountain car scenario 178

Chapter 1

Introduction

This thesis describes research on a new learning paradigm for software agents: learning how to perform a task by observing the actions executed by an expert agent with similar characteristics while performing a task. Learning by observation is one of the least common and most complex forms of learning amongst animals because it is singular to humans and to a strict number of superior mammals (Bandura, 1977). It is an efficient mechanism of social learning and social interactions of both humans and a number of other animal species, as well as for artificial systems. This method of learning is also usually known in the artificial intelligence community as imitation learning, learning from demonstration, programming by demonstration, learning by watching and learning by showing. This variety arises from the definitions of what is learning by observation, imitation and other similar activities. For consistency, the expression learning by observation is going to be used from here on.

Explaining the ability to learn by observing in humans and other animals has proved to be a complex subject. The mechanisms that underlie learning by observation are not yet quite known and in addition they are closely linked to other social aspects such as socialization and communication (Nehaniv & Dautenhahn, 2004). The abilities involved in learning by observation play an important role in several aspects of social development such as communication, interaction and culture.

Traditional research on learning by observation only involved areas such as psychology, biology and ethnology. However, it has been recognised that the synthesis of artificial systems that can be taught by observation, rather than by explicit programming, can also provide important insights on the mechanisms behind learning by observation. Expanding the research on learning by observation to artificial systems allows a better understanding of its nature and ori-

gins, besides providing a better understanding of social intelligence (Nehaniv & Dautenhahn, 2004).

Over the last decades, the subject of learning by observation has come to light in several areas of artificial intelligence such as programming by demonstration, machine learning and robotics. Chapter 2 shows that the vast majority of approaches that most accurately follow the principles of learning by observation defined in Bandura's social learning theory (Bandura, 1977) come from robotics, because robots can directly observe each other. Software systems cannot observe one another in the same way tangible entities can be observed and therefore they usually focus on the learning algorithm, ignoring important issues such as how the knowledge is collected and how software systems observe one another. With the exception of one approach (Machado, 2006; Machado & Botelho, 2006), software systems usually disregard that actions may not have visible effects in the environment and do not take into account the direct observation of the activities performed by software systems. The research described in this thesis addresses the following problems:

- How to make software agents observable to themselves and to others.
- How to provide learning by observation abilities to software agents, specifically the ability to learn control knowledge by observing the actions performed by expert agents.

To avoid misconceptions, the act of observing a software agent is defined as:

> Reading meta-data about an agent's constituents, its actions and the conditions holding when the actions were executed, without a direct intervention of the observed agent.

This way, the observed agent (the expert) has a passive role in the observation process, as would be expected in learning by observation in humans and superior mammals. It is the agent that wants to learn (the apprentice) who takes action to obtain new knowledge without interfering with the expert agent.

The following sections present the motivation for this research (section 1.1), an overall view of the approach to learning by observation (section 1.2), the important aspects to retain from the evaluation of the approach (section 1.3) and a description of the application scenarios (section 1.4).

1.1 Motivation

This section describes the importance of learning by observation for software agents and the conditions under which learning by observation is an adequate method for software agents to learn. Research in neurology and psychology shows that learning by observation may well be one of the causes of the exponential growth of human technologies in the last centuries. Unlike natural selection, observation allows these capabilities to spread amongst individuals within the same generation (Ramachandran, 2003). Learning by observation allows humans and animals to learn faster than when using other learning methods (Meunier *et al.*, 2007).

Machine learning approaches usually disregard the process of discovering experts and collecting the examples that are necessary for learning a task. However, when compared with learning by observation in humans, such a learning model seems unnatural. In learning by observation the apprentice has an active role in discovering experts and collecting the examples. The learning process, that takes place in the apprentice, includes the discovery and comparison of the structures of the experts to determine their similarity with the apprentice (Bandura, 1977; Ramachandran, 2003; Meunier *et al.*, 2007).

When applied to artificial systems, learning by observation allows artificial agents to share skills without the need of human intervention. Apprentices can automatically learn about the expert's behaviour and include it in their own repertoire. The use of learning by observation also broadens the context of the interactions and collaboration between artificial systems and humans. It provides the means for the establishment of social relationships between apprentices and experts (Nehaniv & Dautenhahn, 2004). For example, learning by observation could be used as a means of establishing a social relationship between a personal agent and its user. This would allow the personal agent to be aware of the actions of the user and thereby learn how and when to perform them.

Learning by observation is useful for software systems because it provides a more direct approach to the problem to solve when compared with other techniques where agents learn only from the transitions in the state of the world. Instead of spending time testing several hypotheses, artificial agents acquire the knowledge from the expert by directly observing its actions while it performs the task. This allows the artificial agents to directly know which actions are necessary to perform a specific task (Argall *et al.*, 2009; Chernova, 2009).

According to several authors (Byrne, 1999; Dautenhahn & Nehaniv, 2002; Botelho & Figueiredo, 2004; Machado, 2006), being able to observe an expert agent performing its actions, as opposed

to merely rely on the observation of their effects, can be advantageous because not every action produces visible changes on the environment (for example, agent communication). In addition, when the representation of world states requires too much memory it is impossible for the agent to know the effects of all actions (for example, simulations with multiple interactions). The agent may also not be able to know the effects of its actions *a priori* (for example, executing or invoking a software program or API). Observing the actions is also advantageous in situations in which the same effects could be achieved by different alternative actions but using one of them is clearly better than using others (for example, using a set of sums instead of a simpler multiplication). In such cases, it is important to actually ”see” what the expert agent is doing by observing its actions. This allows the agent to know exactly what actions were executed.

The application of learning by observation in software systems is ideal for societies where agents share common features but have their own internal representation methods (for example, integration of legacy systems). Without common internal representation methods, directly transferring knowledge between agents is impossible or, at least very difficult. Learning by observation makes it possible to learn without the need of common internal representations because each agent makes its own interpretations of what it is observing.

The advantages of learning by observation provide the motivation for further research on how to implement a learning mechanism, for software agent societies, that enables agents to learn from each other. The following section provides an overview of the proposed approach to learning by observation in software agents.

1.2 Approach Definition

The research described in this thesis presents a computational approach to enable software agents to learn by direct observation of the actions being performed by expert agents while they are executing a task. An overview of this research is presented in (Costa & Botelho, 2013). This is possible because the proposed approach also concerns displaying information that is essential for observation, making it possible for software agents to observe each other. It defines a visible representation of the agent constituents and capabilities, of the actions it executes and of the conditions holding before the actions were executed - the software image (see chapter 3).

The proposed software image is an improvement of Machado’s approach (Machado, 2006; Machado & Botelho, 2006). It represents the agent as a collection of parts with sensing and

action capabilities and displays snapshots of the agent's current and past activity (see section 1.2.1). It also defines an ontology to provide a common ground for designating the elements of the software image (see section 3.3). Other innovations include a shared repository where agents can register their software image so that it is available to any interested agent and the possibility of observers subscribing to notifications on the creation of new snapshots (see section 3.4.3).

In addition to the software image, the proposed approach presents a learning process that covers all aspects of learning by observation, such as discovering and observing experts, storing the acquired data, learning from the observed information, applying the acquired knowledge and evaluating the agent's progress (see section 1.2.2). Agents can find themselves in one of two states of the learning process: the learning state or the execution state (see chapter 4).

When agents are in the learning state, their only objective is to observe experts and acquire new knowledge from them. When agents are in the execution state, their only objective is to apply the acquired knowledge to select the actions to execute. The agent's evaluation controls switching between these two states (see section 4.6).

Two distinct methods of learning from the information acquired by observation, the recall and the classification methods, were specifically developed for the approach (see section 4.5). The two methods allow agents to perform the observed task when facing the same conditions as the experts and also a similar task when facing different conditions.

The proposed approach is also capable of mimicking the behaviour of mirror neurons, which allows the agent to use the learning methods both when observing and when preparing to execute its own actions (see section 4.1). Another innovation of the proposed approach is the ability to receive external feedback on the executed actions from specialized evaluators.

The following sections provide additional details on the two main aspects of the proposed approach: the visible representation of software agents and the computational approach to learning by observation.

1.2.1 The Visible Representation of the Software Agent

For software agents to learn by observing each other they need to be visible to themselves and to others. To become "visible" to others, the software agent needs an accessible representation of its constituents and activities, which is called "the visible software image" or merely "the software image". This "visible" representation must include at least the agent components and

the actions it performs (Botelho & Figueiredo, 2004).

To make the software agent ”visible” to others, the proposed software image provides a domain-independent meta-data representation of its constituents, of the actions it executes and the conditions holding before the actions were executed. It improves Machado’s proposal (Machado, 2006; Machado & Botelho, 2006) by including the agent’s input mechanisms (the sensors) in the description of the agent constituents (see section 3.2.1) and enhancing the information on the agent actions with the conditions holding when they were executed and with the history record. It also enables the representation of simple and composite agent actions (see section 3.2.2).

The proposed software image defines a meta-ontology to provide a common ground for designating the elements of the software image (see section 3.3). A shared repository was developed to allow agents to register their software image so that it is available to any interested agent. The proposed software image also opens the possibility of observers subscribing to notifications on the executed actions (see section 3.4.3).

Following Machado’s proposal (Machado, 2006; Machado & Botelho, 2006), the information provided by the proposed software image is divided in two main categories: the static image and the dynamic image. The static image is a representation of the agent’s constituents, their relations and their characteristics. The dynamic image is a representation of the agent’s current and past actions and the conditions holding before those actions were executed.

The static image represents the agent as a collection of parts with sensing and action capabilities. The agent parts can also have visible attributes which represent the important aspects of the agent’s internal state. The description of the agent sensors in the static image allows the agents to compare with others not only by the actions they can perform but also by the kind of information they can obtain from the environment. The information collected by the agent sensors represents the way the agent understands the world - the agent’s perspective of the world. Without this information agents have no way of knowing if the experts they observe understand the world in the same way as them, which is important for a better understanding of the reasons behind the expert’s actions (Bandura, 1977; Ramachandran, 2000).

Including the agent sensors in the static image is essential when software agents only have access to part of the state of the environment, which is acquired by their sensors, because it provides the interested agents with information on the agent’s sensing abilities. An example of this is when two agents have access to different properties of the same object making it harder,

if not impossible, to learn from each other because they would not understand the perspective of the other. For example, one of the agents only sees the colour of the object and the other only sees the shape of that object. The information regarding the agent sensors is also an important aspect of embodiment because it is one of the requirements to be situated (Etzioni, 1993).

Unlike Machado's proposal (Machado, 2006; Machado & Botelho, 2006), where only the action being currently performed was displayed, the proposed software image displays snapshots of the agent's current and past activity in the dynamic image (see section 3.2.2). The snapshot captures all the agent's activity in the time it takes to update the agent's perception of the world. It holds a representation of the executed actions and of the conditions holding at the moment the agent decided to execute them. The conditions represent the state of the environment and important aspects of the agent's internal state at a given moment. The information provided for observation (the sequence of snapshots) is similar to the training sequences used for training machine learning algorithms (a sequence of condition-action pairs) which facilitates the development of the learning algorithm (see section 4.5).

The conditions in the snapshot can either hold the perspective of the agent that executes the actions (the observed agent) or the perspective of the agent that is observing (the observer agent) (see section 3.1). Using the perspective of the observed agent requires the observer to have the same kind of sensors and visible attributes as the observed agent because it is the only way it can understand the information contained in the conditions.

Using the perspective of the observer does not require both agents to have the same sensors. However, it requires the observer to be in sync with the observed agent because the observer needs to know exactly when the observed agent acquires the conditions for the actions it executes. In dynamic environments it is not guaranteed that the conditions holding before the actions are visible in the dynamic image are the same as the conditions holding at the moment the agent decided to execute them. In addition, it is not always ensured that the observer has a complete access to the environment of the observed agent. When agents run in distinct processes and have no skills to communicate with other processes, they can only acquire the state of the environment that is contained in their process.

In the proposed approach, the conditions hold the perspective of the agent that executes the actions because it is essential for the history record (see section 3.2.2). The history record provides access to the agent's past actions and the conditions holding when those actions were executed in an uninterrupted flow from a given moment in the past to the present moment. This

allows apprentices to acquire a large amount of information without waiting for the expert to perform the actions. Without the perspective of the agent that executes the actions it would not be possible to store the conditions whenever the agent is not being observed.

The dynamic image also enables the representation of simple and composite actions in the snapshots. A composite action is composed of a sequence of actions which reproduce a complex behaviour. Instead of displaying this complex behaviour as a single action, the behaviour is decomposed in simpler actions which are easier to learn. The simpler actions may also be reused to create other complex behaviours (see section 3.2.2).

The proposed software image uses an ontology to describe the knowledge about the agent sensors, visible attributes and actions, about the tasks to be accomplished and about the relationships between these elements. Using ontologies allows different agents that follow the same ontology to use the same designations for the same kinds of sensors, actions and visible attributes and for the same tasks. The ontology also enables specific kinds of sensors, actions and visible attributes to be associated with specific tasks, which allows the agents to know which elements are required to perform a task (see section 3.3).

Another important aspect of the ontology is the possibility of associating different designations of the same element, which opens the possibility of translations between different ontologies. A meta-ontology, which is called the software image meta-ontology, was created to facilitate this translation. The meta-ontology defines the basic elements of the ontology and the possible relationships between those elements.

A shared repository was specifically developed for the proposed software image (see section 3.4.3). Software agents use this repository to register their software image. The registered software images can be easily accessed by any agent interested in observing the represented agent. This ensures that, even though running in different processes, the agents can still observe each other.

The proposed software image also provides a notification service that allows observer agents to subscribe to snapshot notifications. The subscribed observers receive notifications each time a new snapshot is created in the dynamic image of the observed agent. This allows the observers to know exactly when they have to observe, that is, when they can collect a new snapshot from the dynamic image of the observed agent (see section 3.4.3).

The following section shows the relevant aspects of the approach to learning by observation.

1.2.2 Learning by Observation of the Software Image

The proposed approach defines a complete learning process which includes discovering an expert from which it is potentially possible to learn, observing that expert, retaining the acquired information in the agent's memory, learning new control knowledge from the information in the agent's memory to propose actions, applying the acquired knowledge and a continuous evaluation of the agent's progress (see chapter 4). The proposed learning process follows the definition of learning by observation provided by the social sciences (see section 2.2.1), with some limitations regarding the motivation that drives an apprentice to observe an expert, which is only partially covered by the proposed approach.

The motivation to observe an expert is addressed by the internal evaluation which provides the agent with a measure of the confidence on its knowledge. Depending on this confidence, the agent can be motivated to acquire more knowledge (when it stops being confident) or to perform the task using the acquired knowledge (when it is confident on its knowledge). The agent can therefore find itself in one of two states of the learning process: the learning state or the execution state (see chapter 4).

When agents are in the learning state, their only objective is to observe experts and acquire new knowledge from them. When agents are in the execution state, their only objective is to apply the acquired knowledge to select the actions to execute. Switching between these two states depends on two configurable thresholds for the confidence. The agent can also switch to the learning state whenever it faces a configurable amount of unfamiliar conditions, that is, conditions that were not faced by any of the observed experts (see section 4.6).

The proposed approach mimics the behaviour of the mirror neurons by allowing the agent to learn new control knowledge and propose actions when it is preparing to execute its own actions and also when it is observing (in the learning state). This is possible because there is a distinction between the representation of an action and its execution. The agent proposes abstract representations of its actions, meaning that they are not immediately executed.

Being able to propose actions when observing and when preparing to execute them not only improves the identification of the observed actions but also allows the agent to evaluate its knowledge while it is learning (see section 4.6). When observing, the agent proposes actions for the conditions in the observed snapshots, that is, for the conditions faced by the observed expert. The proposed actions can then be compared with the observed actions (the actions in the snapshot) to determine if the agent is able to propose the same actions as the observed expert.

The proposed approach defines two methods of determining the experts from which it is potentially possible to learn. Bandura's social learning theory (Bandura, 1977) is followed when it is not possible to determine the agent structures are necessary for the task to learn. According to this theory, agents can observe experts that are similar to them, that is, an expert whose static image has the same structure and the same elements as the agent's static image (see section 3.4.3).

When it is possible to determine the task to learn and the structures and abilities that are necessary for that task (see section 3.3), agents can observe an expert as long as the intersection of their software images contains those structures and abilities. This is enough to ensure that the apprentice agent is able to recognize all the conditions and activities on the expert software image that are necessary for learning a specific task.

Before acquiring the snapshots regarding the expert's current activity, the agent acquires all the snapshots in the expert's history record. This provides a large amount of knowledge without waiting for the expert to execute those actions. The process of discovering an expert and observing it is referred to as the observation period. The agent may go through several observation periods, where it has the chance to observe different experts, while it is learning. This increases the diversity of the agent's knowledge because different experts might provide different points of view on the task to learn (see section 4.3).

The observed snapshots are retained in a temporary memory before being processed and stored in the agent's memory. The temporary memory acts as a buffer for observation because it allows the agent to handle snapshots at a different rate from which they are acquired. It also allows the agent to keep a record of the expert's actions that might take place while it is acquiring the expert's history, which provides an uninterrupted sequence of snapshots from a moment in the past until the current moment.

In the proposed approach, the agent's memory is stored in a tree structure. Using a tree structure enables the sequence in which the snapshots were observed to be intrinsically preserved in the agent's memory. The tree structure also facilitates the consolidation of the agent's knowledge because it stores different approaches for the same task as alternative paths in the tree (see section 4.4).

The information in the agent's memory is used for learning new control knowledge and proposing actions. The proposed approach provides two methods of learning from the information acquired in observation: the recall and the classification methods (see section 4.5). The

recall method was totally developed for the proposed approach. It uses the sequence on which the expert actions were observed to propose actions for the new problems. The classification algorithm is an adaptation of the existing KStar algorithm (Cleary & Trigg, 1995). It categorizes the information in the observed data and determines which actions to perform according to the categories of the new problems.

The two methods propose actions individually, providing the agent with a broader set of possibilities of actions to choose from. Dynamically computed weight factors are associated to each method to assist on deciding which of them is best suited for the current situation. The weight factors reflect the amount of correct and incorrect actions proposed by each method throughout time, as explained in section 4.5. The accuracy of the proposed actions is determined by the evaluation of the agent's progress, which is described in section 4.6.

The agent's evaluation constantly tests the agent's knowledge, when it is in the learning state and when it is in the execution state (see section 4.6). The results of those tests update the value of the agent's internal confidence. When the agent is in the learning state, the internal evaluation determines if the agent masters the task it is observing. It proposes actions for the conditions in the observed snapshot and checks if the proposed actions are the same as the ones observed in the expert. If the actions are the same the confidence increases, if they are not the same the confidence decreases (see section 4.6).

When the agent is in the execution stage, the internal evaluation determines the quality of the actions selected for execution. A simple monitoring of the executed actions checks whether the actions were effectively executed or not. The confidence decreases if a problem is detected when executing the actions. However, this simple monitoring has some limitations because even though there are no problems when executing the actions it is not possible to ensure that the action was the most appropriate for the current conditions.

To overcome this problem, the proposed approach enables the evaluation to receive external feedback on the executed actions from specialized experts, the teachers, or from other evaluators that are specific to the application domain. This allows the agent to be evaluated on its ability to propose the correct actions for the current conditions. The feedbacks can either be positive, when the agent executes the correct actions, or negative, when the agent proposes the incorrect actions. However, the ability to receive external feedback approximates the approach to the paradigm of learning by teaching.

Another improvement of the external feedbacks comes from the negative feedbacks. A neg-

ative feedback represents a situation where the action proposed by the agent is inappropriate for the current conditions, which is also a negative training example. Therefore, besides influencing the agent's confidence in its knowledge, the external feedbacks also provide the agent with negative training examples. The negative training examples help to prevent future executions of the same actions under the same conditions as explained in section 4.6.

1.2.3 Approach Contributions and Limitations

The proposed approach heavily relies on an agent architecture with software image. Agents with software image can be ”seen” (consulted) by other agents and even by themselves. In spite of the similarities with Machado's approach (Machado, 2006; Machado & Botelho, 2006), the proposed approach to the software image introduces several improvements, in particular:

- The inclusion of the agent sensors in the static image, in addition to the visible attributes, the actuators and the actions from Machado's approach. This improvement provides a better description of the agent and improves the observer's ability to understand the reasons for the agent actions, as explained in section 3.1.

- The dynamic image, besides displaying the executed action as in Machado's approach, also displays information on the conditions holding at the time the action was executed, that is, the state of the environment, as perceived by the agent sensors, and the instances of the important aspects of the agent's internal state. With this improvement, the data provided for observation is similar to the training sequences used for training machine learning algorithms (condition-action pairs), as explained in section 3.2.

- The proposed software image displays historical information about the actions executed in the past and about the conditions holding at the time those actions were executed. This enables the agents to acquire a large amount of knowledge at the beginning of the observation, as section 3.2.2 shows.

- The proposed software image uses ontologies to hold the knowledge on the designations of the different kinds of sensors, visible attributes, actions and tasks. This improvement allows agents that follow the same ontology to use the same designations for the same kinds of sensors, visible attributes, actions and tasks, as section 3.3 shows.

- The proposed software image provides a shared repository where agents can register their software image. The registered software images can be easily accessed by any agent interested in observing the represented agent, even when running in a different process (see section 3.4.3).

- The proposed software image allows observers to subscribe to snapshot notifications. This allows the subscribed observers to know exactly when to observe, which is when a new snapshot is created in the dynamic image of the observed agent (see section 3.4.3).

The contributions of the proposed approach are not restricted to the agent software image. The other important contributions are the complete approach to learning by observation in software agents, in particular:

- The discovery of experts which provides the agent with the necessary tools to discover, by itself, expert agents from which it is possible to learn (see section 4.3).

- The use of tree structures to hold the agent's knowledge, which allows the agent to express its knowledge as a decision tree and enables storing different approaches to the same task with minimal conflicts (see section 4.4).

- The definition of two learning algorithms, which are used to convert the observed information into mechanisms for choosing the actions to perform in the current conditions - the recall algorithm and the classification algorithm. The recall algorithm was totally developed for the proposed approach and uses the sequence on which the expert actions were observed to propose actions. The classification algorithm is an adaptation of the existing KStar algorithm (Cleary & Trigg, 1995) and categorizes the observed data to determine which actions to propose according to the categories of the new problems (see section 4.5).

- The definition of an internal evaluation process which provides the agent with a measure of the confidence on its knowledge. Depending on this confidence, the agent can be in one of two states of the learning process: the learning state or the execution state (see chapter 4). When agents are in the learning state, their only objective is to observe experts and acquire new knowledge from them. When agents are in the execution state, their only objective is to perform their task using the acquired knowledge. Switching between these two states depends on two configurable thresholds, as explained in section 4.6.

- The ability to mimic the mirror neurons, which allows the agent to use the same mechanisms to propose actions both when learning and when using the acquired knowledge. This allows the agent to directly associate the observed actions with its own actions. It also allows the agent to propose actions for the conditions faced by the observed expert and determine if the agent is capable of proposing the same action as the observed expert (see section 4.1).

Despite the contributions to advance the state of the art of learning by observation in software agents, this approach has a limitation regarding the discovery of experts from which it is possible to learn. This limitation was also found on the surveyed approaches for learning by observation (see section 2.2.5). In normal circumstances, although an expert has the same features as the apprentice agent, it does not necessarily mean that it is performing the actions that are necessary for the apprentice to learn (by observation) how to perform a specific task.

As in almost all the surveyed approaches, the experts in the demonstration scenarios (see chapter 5) are prepared to only execute the actions that are necessary for the task to be learnt. One possibility of overcoming this limitation would be through communication with the observed experts, that is, the apprentices could request the expert to perform a specific task. However this possibility would require the definition of a communication protocol between the observer and the expert, which, given the time limitations, is outside of the scope of this thesis.

1.3 Evaluation

The main objective of the research is the development of an approach to learning by observation that allows software agents to learn control knowledge by observing the actions of another software agent while it is performing a task. The research is evaluated by the following criteria:

1. A learning by observation agent must be capable of discovering the correct experts from which it can potentially learn by comparing its software image with the software images of the expert agents.

2. A learning by observation agent must be capable of solving a problem after observing an expert solving that same problem.

3. A learning by observation agent must be capable of solving a problem after observing experts solving the same problem in different contexts.

4. After learning, the agent's performance, in terms of the number of actions correctly executed, must be similar to the performance of the observed experts.

5. A learning by observation agent must improve its ability to master a task each time it goes through an additional learning period.

6. The learning by observation agent must obtain better results, in terms of the ability to master a task and the time it takes to learn that task, than other learning mechanisms under similar circumstances.

The secondary objective of the research is to provide software agents with a universally accessible visible representation. Ideally, this visible representation, the software image, would fulfil all possible goals regarding the visible (accessible) representation of software agents. Unfortunately this research is far from providing useful insights about all possible uses of the software image. The proposed visible representation expands the former approach by Machado (Machado, 2006; Machado & Botelho, 2006) with new features suitable for learning control knowledge by observation. The proposed approach also has in mind that it should be as general as possible not impairing future uses.

The proposed approach was implemented in three different scenarios that show the capabilities of the approach and validate the evaluation criteria. The following section presents an overview of those scenarios.

1.4 Demonstration Scenarios and Results

This section presents a general description of the demonstration scenarios where the proposed approach was implemented. The capabilities of the proposed approach are tested in three different scenarios, described in chapter 5. In all the demonstration scenarios, the agents do not initially know the effects of their actions and, with the exception of one setting in the first scenario, no external feedback is used to enhance the agent's evaluation (see section 4.6).

In the first scenario, the agent learns how to manipulate their virtual hand to display numbers using sign language (see section 5.2). In the second scenario, the agent learns how to calculate mathematical expressions using the numbers and operators acquired from the environment (see

section 5.3). In the third scenario, the agent learns how to reach the top of a simulated mountain by going backwards and forwards to gain momentum (see section 5.4).

In addition, chapter 5 also presents the results of the tests on the proposed software image (see section 5.1). The results show that the agent's complexity can be measured by the number of parts, the number of atomic elements in those parts and the depth of sub-parts. Agents composed of a small number of parts with a small number of elements and depth are less complex and therefore have the shortest times both when creating and when comparing their software images. The tests also show that this measure of complexity only affects the time to discover an expert from which it is potentially possible to learn. The time it takes to acquire a snapshot from the dynamic image is not affected by the agent's complexity.

The first scenario (see section 5.2) was especially conceived as a situation in which the results of agent actions are not visible in the environment, since the majority of the actions only affect the agent. Therefore, it will not be possible for a common machine learning solution to learn only from the effects of the actions. For a complete understanding of what is happening, it is necessary to observe the agent's constituents and its actions.

The scenario simulates an agent with a virtual hand that displays numbers using sign language. Each time the agent sees a number, the virtual hand is changed to display that number by means of sign language. The virtual hand is used only for communicating with users through a graphical display. The only way software agents can obtain information on the state of the agent's virtual hand is through the software image.

The first scenario was specially conceived to test the abilities of the proposed approach and the influence of the configurable thresholds. It shows that the confidence thresholds determine the time the agent spends learning and the number of times it returns to the learning state after starting to execute its own actions, which affects the total amount of executed actions. The configurable amount of unfamiliar conditions affects the number of times the agent returns to the learning state after starting to execute its own actions, which also affects the total amount of actions executed.

The scenario also determines the influence of the two methods of proposing actions, recall and classification, on the executed actions. The recall method has more influence when the agent faces the same conditions as the experts it observes. The classification method has more influence when the agent faces different conditions (see section 5.2.1). Finally the scenario shows that using external feedback from a specialized expert is useful when the effects of the

actions are not visible both in the environment and in the agent (see section 5.2.4).

The second scenario (see section 5.3) compares the proposed approach with supervised learning algorithms that only learn from the effects of the actions, in a situation where some of the effects of the actions are not visible in the environment. The scenario was conceived to show the inefficiency of only observing the effects of the actions under such conditions. The agents have to learn how to calculate mathematical expressions from the numbers and operations acquired from the environment.

The agents hold the acquired numbers and operators to create a mathematical expression. The expression is calculated and the agent outputs the result of the calculation. The effects of the actions concerning obtaining the numbers and operators to create the expression are partially visible in the environment. The part of the effect that is visible is their acquisition from the environment. The other effect of those actions, which is calculating the value of the expression, is not visible because it only affects the agent. The only action whose effect is completely visible in the environment is the output of the result.

The third scenario (see section 5.4) presents the mountain car problem as described in (Sutton & Barto, 1998). It is an application of the proposed approach in a different class of problems, which are are best suited for reinforcement learning (RL) algorithms. The proposed approach is compared with a RL algorithm in a situation in which this kind of learning algorithm has already shown to be a good approach. (Mitchell, 1997).

The outcome of the tests performed in the application scenarios validates the evaluation of this research (see section 5.5). The results show that, with the proposed approach, agents that do not initially know the effects of their actions can correctly learn how to perform a task after observing experts performing that task. The agents are also able to perform a task after observing experts performing similar tasks under different conditions (see section 5.2.1).

The first scenario also shows that agents are capable of returning to the learning state after starting to apply their knowledge (the execution state) and increase their ability to perform a task each time they return from the learning state. Finally the scenario shows that even when the effects are not visible in the agent (see section 5.2.2), it is still possible to learn the task, even though it requires additional feedback from specialized experts (see section 5.2.4).

The second scenario (see section 5.3) shows that observing the actions is better than observing their effects in situations where some of the effects of those actions are not visible in the environment. Even when compared with a KStar algorithm (which is used by the classification

method) that only observes the effects of the actions, the approach to learning by observation is far better when considering the amount of correct actions executed. This shows the advantages of the approach when compared with similar solutions that only observe changes in the environment.

The third scenario (see section 5.4) shows that the proposed approach is able to learn faster than the reinforcement learning approach even in problems for which reinforcement learning has proved to be adequate. Agents using the proposed approach are able to learn faster and require fewer actions to achieve the goal. They are also able to achieve approximately the same results as the experts (in terms of the number of actions and the distance that is necessary to travel to reach the top of the mountain) which is far better than the results of the reinforcement learning apprentices.

Even though learning by observation requires the presence of experts, in typical learning by observation applications the experts already exist. Learning by observation does not require an additional effort developing expert knowledge. The only situation in reinforcement learning where there is no effort in developing expert knowledge is rewarding the agent only at the end of the task. The only information provided is the goal, which is embedded in the agent. The application of any other reinforcement schemes would require an additional effort on developing a proper reward function or on directly providing the rewards for the performed actions.

In the proposed approach, agents have the ability to find experts by themselves (see section 4.3.1). These experts can either be other agents, with the sufficient knowledge for the task, or even a program that is directly controlled by a person. The ability to find an expert removes the burden of building a proper reward scheme which would be necessary to improve the reinforcement learning agent. Therefore, the proposed approach is more advantageous than reinforcement learning since it provides shorter learning periods, better results and even a smaller development effort.

The next chapter presents a survey on research on the visual representation of agents and on learning by observation. Chapter 3 describes the approach to the software image. Chapter 4 describes the approach to the learning by observation process. Chapter 5 describes the experimental results and the evaluation of this research. Finally, chapter 6 presents conclusions and future work.

Chapter 2

State of the Art

The main contribution of this thesis is a new learning paradigm for software agents: learning how to perform a task by observing the actions executed by an expert agent with similar characteristics. The main intention of this chapter is to survey the approaches that are related to this learning paradigm or that may contribute to solve the problems faced when applying learning by observation in software agents. With this survey it is possible to determine the relevant features for an approach to learning by observation and what is still lacking from the surveyed approaches.

Learning by observation can be, in certain domains, the most efficient learning method, but as this survey shows (see section 2.2.5), until now the major advances concerning the ability to directly observe one another are related to robotics. The main reason for this is because robots can physically observe each other. They also allow a more natural interaction with humans which facilitates using human experts and enables the creation of an intuitive communication medium between humans and robots (Argall *et al.*, 2009).

Software agents do not have the ability to see themselves or the others in the same way as tangible entities (such as robots) see themselves and other tangible entities. This is the main reason why they are usually constrained to learning by observing the effects of actions, instead of the actions themselves (Quick *et al.*, 2000; Argall *et al.*, 2009).

However, as several authors have already emphasized (Byrne, 1999; Dautenhahn & Nehaniv, 2002; Botelho & Figueiredo, 2004; Machado, 2006), not every observable action produces visible changes in the environment. Thus, using state change information alone in learning algorithms is not always a good option. In such cases, it is important to actually ”see” the expert agent executing its actions. This approach allows the apprentice agent to know what

action was performed by the expert, thus overcoming the problems that arise when the effects of the actions are not visible.

For software agents to learn by observing the actions of other software agents, the existence of a visible body that displays the necessary visible features is mandatory (Mataric, 1997; Botelho & Figueiredo, 2004; Machado, 2006). Software agents need to be equipped with some kind of visible representation of their components and of the actions they do. To make a software agent "visible" to others, its visible representation must be accessible not only to its owner but also to other agents.

Determining how to create this representation of the software body, which is called the software image (of the visible software agent), how to make it universal and how observation and interpretation takes place, are important aspects of this thesis. Therefore, the two main contributions of this chapter are:

- The literature overview regarding approaches for building a visual image for software agents (Section 2.1)
- The literature overview regarding learning by observation which includes the social learning theories (Section 2.2)

The literature overview on embodiment, embodied cognition and embodied artificial intelligence in section 2.1.1 shows that embodiment deals with problems that are similar to the representation of a visible software image. However, the main aspects focused by the embodiment literature are the distinction between mind and body, the grounding problem, the advantages of having a body and the definition of what is embodiment. This leaves little room for approaches that address more specific aspects of embodiment (such as how to represent a software body) and that can point out to different ways of solving this problem.

It is also possible to find contributions for the visible representation of software agents and their actions outside the embodiment literature. As section 2.1.2 shows, the approaches for the graphical representation of software programs and of the interactions between their components may contribute with a different perspective on how to build this visible representation, even though it is not their main purpose.

From what is acknowledged by this literature revision, the only known approach to have proposed a software image for visible software agents comes from Machado (Machado, 2006; Machado & Botelho, 2006). With this software image, software agents are able to show their

constituents and the actions they are performing. However, as section 2.1.6 shows, the mentioned approach misses some important requirements such as the inclusion of the agent sensors in the description of its constituents and capabilities, the ability to represent complex actions, the existence of a history record, the inclusion of the conditions in the data provided for observation, the inclusion of the agent's visible properties as conditions for the executed actions, the existence of a common repository for the software images and the possibility of receiving notifications on agent activities.

The literature overview of approaches for learning by observation described in section 2.2 shows that the learning algorithm is one of the most important aspects of an approach to learning by observation (see section 2.2.2). However, an approach cannot be limited to the definition of the learning algorithm. It must also provide a global view of the learning process that includes the agent's motivation to learn, discovering and observing agents, learning new knowledge from the information acquired by observation, storing this knowledge and applying the learnt knowledge in new situations (Demiris & Hayes, 2002; Tan, 2012). Evaluation is also an important aspect to consider in the learning process (Wood, 2008; Billing *et al.*, 2010).

One of the major flaws detected on the surveyed approaches, besides lacking the ability to directly observe the expert performing the task, was the fact that this global view is still missing (Tan, 2012). To the best of our knowledge, with the exception of the proposed approach, all approaches are focused on solving specific problems, and the solutions they provide are supported exclusively by the learning algorithm, which is only part of the learning process. The learning algorithm defines how the information, obtained from observation, is stored and how it is used, that is, how the agent proposes actions to execute when facing new problems (Argall *et al.*, 2009).

Section 2.2.2 shows the different possibilities for the learning algorithm. The most common possibility is to follow the same sequence of actions as the expert, or sequencing, which requires the agent to store the sequence on which it has observed the actions performed by the expert. Other possibilities are to generalize the acquired information, to use analogies between the acquired information and the new problems or to categorize the information and determine to which set of categories the new problems belong. This allows the agent to face situations that have never been observed, because in a real world situation it is almost always impossible to observe all possible conditions (Argall *et al.*, 2009; Sullivan, 2011). These possibilities may use Bayesian or classification algorithms, neural networks, inverse reinforcement learning, case-

based reasoning or inductive programming.

This variety of possibilities for the learning algorithm reinforces the remark that learning by observation does not depend on a specific method or algorithm to create new knowledge from the observed data. However, the literature overview shows that the approaches are somehow dependent on the types of domain on which the chosen algorithms are more effective.

Another important conclusion of the survey on the approaches for learning by observation is described in section 2.2.4. It shows that learning by observation cannot be fully achieved without mimicking specific structures that exist in the brains of humans and superior mammals - the mirror neurons - because they are tightly involved with learning by observation. Several approaches propose to mimic the peculiar behaviour of the mirror neurons through forward models. Forward models allow apprentices to create internal representations of the world, from which they obtain information on the consequences of the actions without executing them.

However, the forward models are specifically designed for robots and use hardware inhibitors to prevent the robot actuators from executing the estimated actions. The main objective of these kind of structures is to create a distinction between the description of the action and its execution, that is, to create abstract representations of agent actions (Kulic *et al.*, 2011). This solution is simpler and provides the agent with control over the execution of the actions proposed by the learning algorithm.

The following sections present an overview of the surveyed approaches on embodiment, on the visual representation of the software agent and on learning by observation.

2.1 The Software Visual Image

This section presents a survey on embodiment, embodied cognition and embodied artificial intelligence, the research fields that are closely related to the subject of providing software agents with an accessible (visible) representation of their constituents (their body) and activities. Without the means to represent the constituents of a software agent and the actions it executes, learning by observing an expert while it performs a task becomes harder, if not impossible, to achieve.

The main purpose of this section is to provide an overall view of what is embodiment and to show how the literature on these research fields can point out important contributions of the software body and how they can be used to provide a ”visible” description of the software

agent and of its activity - the visible software image or simply the software image. The section also shows that outside of the literature on embodiment it is also possible to find approaches that, although not aimed to solve the problem of providing agents with a software image, make some contributions on how to build it. These approaches are usually directed to the graphical representation of programs and of the interactions between their components, for software developers.

The literature on learning by observation also has contributions for the software image. Although the approaches to learning by observation are mainly focused on how to learn from observing the actions of an agent, some of them provide insights on how to represent the agent's activity in a way that is "observable" to other agents. The literature on learning by observation also provides some improvements on the software image, such as the use of ontologies to allow a common understanding of the information contained in the software image.

2.1.1 Embodiment

Embodiment consists of giving the notion of body to an entity. It is supported by the phenomenology school, which believes that the role of the body, in the interaction between agent and environment, should be more significant. Despite the concern with the role of the body and its properties, the literature on embodiment does not care about the definition of a software image.

Claims about embodiment revolve around materiality and the physical world, which makes it difficult to understand how to embody something in a software world (Quick *et al.*, 2000). However, it is possible to highlight some of the features that are important for a "visible" description of the agent constituents such as the agent's acting and sensing capabilities (Brooks, 1991; Etzioni, 1993).

Dreyfus (Dreyfus, 1992) suggests that embodiment needs relatively little emphasis on the actual form of the body, but rather on qualitative features such as the capacity to act, *"the 'I can' ability to respond to situational solicitations"* (Dreyfus, 1992). The possibility of performing actions is not something that is inherited from being material, unless the action is happening in the material world. This makes being a material agent not immediately significant (Quick *et al.*, 2000).

Software agents can also be situated in the software environment in the same way as a robot is situated in the physical world, if characterized by what they can do (their actions) and also

by the kind of inputs they are able to collect from the software environment (Etzioni, 1993). Stripping embodiment from its association to physicality makes it possible to find applications of it for software agents. One of these possible applications is learning by observation, as stated by Botelho and Figueiredo (Botelho & Figueiredo, 2004) and Mataric (Mataric, 1997).

Despite the advancements in research on embodiment, recent approaches for software embodiment are usually associated with the representation of virtual characters, such as described in (Morel & Ach., 2011). Like the robotic approaches, the problem of the existence of a body is overcome by replacing it with a virtual representation of something that exists in the physical world, which in this case is the body of the virtual character.

Additional information on embodiment and embodied cognition is presented in appendix A. The following sections show contributions from outside the embodiment literature on creating the software image.

2.1.2 Code Analysis and Function Tracking

This section shows that besides the literature on embodiment, an overview of other subjects provides ideas that can be used for the representation of the constituents and activities of software agents, or simply, the software image. Code analysis (Consens *et al.*, 1992) and function tracking (Arévalo *et al.*, 2010) are examples of approaches from outside the literature on embodiment that provide ideas on how to represent the constituents and activities of software agents.

Although the main purpose of these two approaches is not directed to solving the problem of creating an accessible representation of the software agent their ideas can be adapted for this representation. Code analysis is normally used to build graph like representations of the constituents of a program. It uses reflection and code introspection tools to build a graphical representation of the program's classes and the relationships among them. It allows developers to figure out how these programs are made (Consens *et al.*, 1992).

Function tracking describes the activities of a program through tracing and logging. It provides detailed reports on the sequence of messages exchanged between the program constituents when carrying out a specific activity. The main purpose of function tracking is to help developers understand the interactions inside a program (Arévalo *et al.*, 2010).

When combined, code analysis and function tracking provide a practical solution on how to describe the agent constituents and to trace its activities. However, the descriptions they provide are low-level representations such as the description of the program classes and methods.

A better approach is to abstract the program constituents and activities which improves the capability of detecting and comparing similar concepts. With this improvement, it would only be necessary to define a mechanism to make these representations accessible (and thus visible) to the agents that might want to observe them.

The following sections show how approaches for learning by observation can contribute to the definition of the software image.

2.1.3 Requirements on the Observed Data

This section shows the contributions from the literature on learning by observation on the kind of information to provide for observation - the demonstration data. The section also shows several approaches for observing, that is, for providing the demonstration data to the agents that want to observe an expert performing a task - the observer agents.

According to Argall and her colleagues (Argall *et al.*, 2009), the most suitable kind data for observation, that is, for the demonstration data, is the condition-action pair, which is similar to the data used for training machine learning algorithms. When agents observe an expert they are actually acquiring a condition-action pair, consisting of the actions executed by the agent and the conditions holding before those actions were executed. The condition-action pair acquired by the observer agent represents a positive training example provided by the observed expert (Argall *et al.*, 2009).

The conditions in the demonstration data represent the way the agent understood the world at the time the actions were selected and executed. The conditions can either be obtained by the agent that is observing, the observer, or by the agent that is performing the actions, the observed expert. When the conditions are obtained by the observer, they hold the point of view of the agent that is observing, which means that it is able to immediately understand them. However, it requires the observer and the expert agent to be synchronized so that the observer knows when to obtain the conditions for the actions it observes (Argall *et al.*, 2009).

The synchronization between the expert and the observer is not necessary when it is the expert agent who obtains the conditions because the expert knows exactly when to obtain them, which is at the moment it selects the actions for execution (Argall *et al.*, 2009). However, there is a risk that the observer does not understand the point of view of the observed expert and therefore it is not able to understand the conditions. Section 2.1.5 shows how this risk can be mitigated and why it is important for the observer to understand the point of view of the

observed expert.

Argall and her colleagues (Argall *et al.*, 2009) also point out some of the strategies used in robotics to provide the demonstration data to the observer agents. However, none of these strategies make reference to an accessible representation of the agent or even to software embodiment, mainly because the agent is a robot and has a physical body. Some of them, like teleoperation, do not even concern the observation of another agent because the demonstration data is provided by direct manipulation of the observer (for example, a person manipulates the arm of the robot to emulate the movements to learn) (Argall *et al.*, 2009). Teleoperation is one of the most commonly used methods of providing demonstration data to observers, mainly because it does not require the observation of another agent (Billing *et al.*, 2010, 2011; Tan, 2012; Sugimoto *et al.*, 2012).

Other methods of providing the demonstration data that require the observation of another agent use specialized sensors to record the executed actions and the conditions holding before those actions were executed (Argall *et al.*, 2009). The observers only have to know where this information is stored and how to collect it. A similar mechanism for recording and presenting the agent's activities is provided in Machado's approach (Machado, 2006; Machado & Botelho, 2006), as explained in section 2.1.6.

In addition to helping to determine the kind of information that is best suited for observation, the literature on learning by observation also provides insight on how to represent agent actions. The following section shows several proposals for representing those actions.

2.1.4 Representing Agent Actions

This section shows several strategies for representing agent actions as a way of decreasing the complexity of the task to learn. One possible strategy is using composite actions, that is, actions composed of a sequence of actions. Through composite actions it is possible to represent a task as a sequence of high-level actions, which in turn are decomposed in sequences of simpler actions which are easier for the agent to learn. The outcome of a composite action is the same as executing all the actions that compose it (Heyes & Ray, 2000; Byrne, 1999; Sullivan, 2011).

Using composite actions is also a way of overcoming the distinction that is usually made between action level and program level learning (Byrne & Russon, 1998). According to Byrne and Russon (Byrne & Russon, 1998) the program level provides a broader description of the task to learn, that is, the task is described by the sequence of composite actions. The action

level reflects a detailed and linear specification of a task, that is, the task is described as the sequences of actions in each composite action.

Wood (Wood, 2008) also uses the distinction of program and action level when describing the tasks to learn. For him, this distinction can be used for generalizing tasks. For example, the task of building a wall may include both building with bricks and building with blocks. At a program level, both tasks would have the same representation, but at an action level some of the actions for building would be different (Wood, 2008). This would, however, require each task to have different composite actions because they would require different sequences of actions to complete the task.

Another possibility of decreasing the complexity of a task is enabling the instantiation of agent actions. Learning actions that may be instantiated is more effective than learning only concrete actions. Once learnt, an action may be applied to several different circumstances (for example, a generic move action that receives the object to move as an argument can be applied to a ball or to a chair by simply changing the argument) (Sullivan, 2011).

A different possibility of decreasing the complexity of the task to learn is presented in Machado's agent architecture (Machado, 2006; Machado & Botelho, 2006). In her proposal, the software agent is represented as a collection of parts, each one specialized in solving part of a complex problem. This way the problem is automatically divided into smaller and simpler problems that are easier to solve and thus requiring less complex behaviours. The approach used in her architecture is similar to the robotic architectures proposed by Brooks (Brooks, 1986).

2.1.5 The Point of View of the Observed Expert

This section shows the importance of understanding the point of view of the agent that executes the actions - the expert. It shows different approaches on how observer agents can understand the point of view of the experts they observe. The importance of understanding the point of view of the observed agent is described by several authors (Ramachandran, 2000; di Pellegrino *et al.*, 1992; Rizzolatti *et al.*, 1996; Maistros & Hayes, 2001). Without this understanding the observer agents are at risk of not knowing the exact reasons for the actions they observe on the expert.

One possibility for understanding the point of view of the observed expert is to use mappings between their concepts of the world and of their actions. With proper mappings it is possible for an observer to correlate the conditions and the actions it observes on the expert with its own

understanding of the world and its actions. Without these mappings learning becomes ineffective because the observer cannot understand the actions it observes in the expert (Alissandrakis *et al.*, 2002).

The mappings can be manually built as proposed by Alissandrakis and his colleagues (Alissandrakis *et al.*, 2002). They use a pre-built library that links instances of the actions and conditions of the expert, with their specific counterparts as they are understood by the observer agent. The observer uses this library to convert the actions and conditions it observes on the expert to the actions and conditions it is able to understand.

Another possibility for understanding the point of view of the observed expert is provided by social learning theories when explaining how learning by observation takes place in humans and superior mammals. According to social learning, this problem can be easily solved by making entities learn from each other only when they share similar bodies and capabilities (Bandura, 1977). The ability to identify similar entities is analogous to the process of matching the static structures of software agents. If both agents have the same static structure there is no need for mappings, since they can immediately recognize the constituents and the abilities provided by these structures on each other.

Recent developments in learning by observation approaches (Sullivan, 2011; Fonooni *et al.*, 2012) provide different alternatives on understanding the point of view of the observed expert. They use ontologies as a way to generalize the agent actions and their interpretation of the world. Describing these elements in an ontology allows them to be recognized by anyone who is familiar with that ontology. The elements can also use the ontology properties to express their relationships with other elements (Fonooni *et al.*, 2012; Sullivan, 2011).

2.1.6 Previous Proposals on the Software Image

This section describes a previous proposal on the software image from Machado (Machado, 2006; Machado & Botelho, 2006). Machado's proposal is a materialization of Botelho and Figueiredo's proposal (Botelho & Figueiredo, 2004) for a component based approach to the agent's body and its visual image. According to them, the agent's body is a set of components, each with its own set of rules to solve particular problems.

In Machado's approach (Machado, 2006; Machado & Botelho, 2006), the ”visible” representation of the agent is composed of a static image and a dynamic image. The static image, the agent's static visual appearance, is made out of representations of the agent's components

and descriptions of their roles. It allows the agent to show its constituents, the relations between its parts and the requirements for their use. The dynamic image, the agent's dynamic visual appearance, represents the agent's activities, body expressions and emotions (Botelho & Figueiredo, 2004; Machado, 2006; Machado & Botelho, 2006).

The static image groups the agent's constituents in parts, each one providing information on a portion of the agent's capabilities. Each capability is represented by an actuator that holds a collection of actions. Each action represents the agent's ability to change the environment or to change its internal state. The agent parts in the static image may also make reference to portions of the agent's internal state - the visual attributes. The visual attributes represent important aspects of the agent's internal state and therefore are made visible in the software image (Machado, 2006; Machado & Botelho, 2006).

The dynamic image needs constant updates because it shows the instance of the action being currently performed by the agent. Unlike the dynamic image, the static image does not change with time to allow a coherent identification among software agents.

Although Machado's approach to the software image (Machado, 2006; Machado & Botelho, 2006) provides a description of the agent constituents and actions, it does not describe the kind of input the agent can acquire from the software environment. As section 2.1.1 shows, for a complete description of the agent it is also necessary to include its sensing capabilities, that is, its sensors. Machado's dynamic image is also only capable of showing the action being currently performed, not the conditions holding for that action. As section 2.1.3 shows, the conditions are also an important aspect to include in the data to provide for observation.

2.2 Learning by Observation

This section provides an overall view of machine learning and how learning by observation fits in the classification of machine learning approaches. It describes how learning by observation is understood on a social and neurological perspective and how this perspective gives insight on determining what experts are appropriate to observe. It provides an overview on the learning algorithms that are most commonly used in learning by observation approaches. It also shows the progress on approaches for learning by observation and identifies specific contributions that might be of key importance for the approach presented in this thesis.

Machine learning is one of the main areas of artificial intelligence. Generally, machine

learning has three major objectives: development of computational theories for machine learning, development of systems with the capability to learn and theoretical analysis and development of generic learning algorithms (Costa & Simões, 2008).

Machine learning algorithms can be organized in supervised, unsupervised and reinforcement learning. Supervised learning consists of creating mappings between inputs and outputs, whose correct values are provided by an expert, which is usually a person. Experts can interact directly with the learner agent providing the necessary training sequences for the learning algorithm (Alpaydin, 2004). Unsupervised learning determines how a dataset is organized by analysing the structure of the input space and identifying which patterns occur more often than others. In unsupervised learning there are no supervisors, there is only input data (Alpaydin, 2004).

Reinforcement learning consists of defining which sequence of actions is optimal for reaching a goal. Expert knowledge is necessary for building a reward system that compensates the agent each time it chooses the correct action. Despite requiring the use of expert knowledge, reinforcement learning is different from standard supervised learning since it does not use training sequences. The outcome of a single action is not important, what matters is the policy, that is, the sequence of actions required to reach a goal (Alpaydin, 2004).

As described in section 2.2.1, learning by observation is an important social interaction mechanism for humans and superior animals from which they benefit from the experiences of others (Alissandrakis *et al.*, 2002). It is the outcome of observing interpreting and reproducing a behaviour model that was executed by others (Billard, 1999; Bandura, 1977). Learning by observation enables cognitive feedbacks and opens the possibility of assigning mental states to others, which are necessary for abstract problem solving such as testing hypothesis, planning in advance, creative thinking and cognitive empathy (Dautenhahn, 1994). The advent of learning by observation in human societies gave way to a new kind of knowledge transference that is embedded by human social interactions (Ramachandran, 2003).

From a computational point of view, learning by observation can be defined as a subset of supervised learning. It comprises all techniques that build knowledge from demonstrated data (Argall *et al.*, 2009; Billard, 1999). A computational approach to learning by observation must also take into account its social dimension (Dautenhahn, 1994; Alissandrakis *et al.*, 2002; Nehaniv & Dautenhahn, 2004; Rao *et al.*, 2004). As section 2.2.1 shows, without this social dimension there is a risk of learning without understanding the reason for the expert's actions

(Alissandrakis *et al.*, 2002). The social dimension of learning by observation also provides an insight on how to determine the kind of expert from which it is possible to learn.

The survey on learning by observation reveals that the definition of the learning algorithm is one of the most important aspects of an approach to learning by observation. The algorithm defines how the information, obtained from observation, is stored and how it is used, that is, how the agent proposes actions to execute when facing new problems (Argall *et al.*, 2009). Section 2.2.2 shows several possibilities for the learning algorithm. One of the most frequently used is sequencing, which essentially is trying to follow the same sequence of actions as the expert. Other possibilities for the learning algorithm are to generalize the acquired information, to use analogies between the acquired information and the new problems or to categorize the acquired information and determine to which set of categories the new problems belong.

The survey on learning by observation also reveals that an approach to learning by observation cannot be limited to the learning algorithm. In addition to the algorithm, the approach must also provide a global view of the learning process, which includes the agent's motivation to learn, discovering and observing experts, learning new knowledge from the information acquired in observation, storing this knowledge and applying it by proposing actions for the current conditions (Demiris & Hayes, 2002; Wood, 2008; Billing *et al.*, 2010; Sullivan, 2011; Tan, 2012). To the best of our knowledge, with the exception of the proposed approach, all approaches are focused on solving specific problems, and the solutions they provide are supported exclusively by the learning algorithm (see section 2.2.5).

The evaluation of the agent's progress is also an important aspect to consider in the learning process because it allows the agent to measure how its performance is affected both when it is consolidating the information acquired from observation and when executing actions (Wood, 2008; Hajimirsadeghi & Ahmadabadi, 2010; Billing *et al.*, 2010; Sullivan, 2011). The evaluation may also provide a simple motivation mechanism because the agent can be motivated to learn when it stops being confident on its performance. The agent can also be motivated to stop learning when it regains the confidence in its performance (see section 2.2.4).

Section 2.2.5 shows that, with the exception of Machado's approach (Machado, 2006; Machado & Botelho, 2006) and of the proposed approach, all major advances in learning by directly observing an agent and the actions it performs are related to robotics (Argall *et al.*, 2009). Software agents, given the problems outlined in section 2.1, cannot directly observe one another. Almost all software approaches for learning by observation are constrained to observe the changes in

the environment (the effects of agent actions) and the knowledge obtained to perform a task is limited to state change information (Quick *et al.*, 2000; Argall *et al.*, 2009).

One of the characteristics of the approaches described in section 2.2.5 is that they usually focus on specific problems, which implies adaptations of some of their contributions whenever they are applied in new domains. Despite these problems, the reviewed approaches provide important contributions that can be adapted to the approach described in this thesis.

2.2.1 Social Learning Theories

This section shows how learning by observation is important for socialization and how it may help to improve agent interaction. It also shows the contributions of social learning theories for a computational approach to learning by observation such as defining what is learning by observation and determining from which experts it is possible to learn.

Research on learning by observation is intimately related to social learning and social interactions, which are frequent in species that live in societies (Dautenhahn, 1994; Billard, 1999). For Meltzoff (Meltzoff & Moore, 1977), learning by observation is a means of probing the person's identity and of communicating with persons, as opposed to things. It is necessary for the *"development of individual contacts, social relationships and a theory of mind"* (Dautenhahn, 1994).

The development of cognitive architectures is the path for achieving a global model for the computational approaches to learning by observation (Tan, 2012). Taking social aspects into consideration when developing computational approaches for communicative agents enables a faster development of common understanding and increases the speed of information transmission (Billard, 1999).

Social learning and learning by observation play an important role in building interpersonal relations throughout child development. The interpersonal relationships allow children to distinguish if something is "like them" or not (Dautenhahn, 1994). The ability to distinguish themselves and to point out similar features in others is important for the identification of entities from which it is potentially possible to learn by observation (Bandura, 1977).

Without the identification of similar features, learning by observation would be impossible because apprentices would not be able to recognize what they were observing on others (Bandura, 1977). Therefore, providing agents with the ability to recognize the structures and abilities of other agents is an important aspect when developing a computational approach to learning

by observation (see section 2.1).

Bandura's social learning theories (Bandura, 1977) are generally accepted as those that best describe the learning by observation process in humans and superior mammals. Bandura emphasizes the importance of observing and modelling the actions of others because when it is not possible to know the effects of those actions *a priori*, learning only from the effects of the actions becomes a laborious process (Bandura, 1977)

For Bandura, learning by observation consists of the organization and symbolic representation of the model of the observed actions (the sequence of actions that are necessary to complete a task) and posterior reproduction of that model. He describes learning by observation as a process with four components: attention, retention, motor reproduction and motivation (Bandura, 1977).

Bandura's description of the learning process gives an insight on the relevant tasks of a computational approach to learning by observation. Demiris and Hayes (Demiris & Hayes, 2002) developed an approach to learning by observation inspired in this description of the learning process. For these authors the learning process consists of three stages: attention (gathering training data through observation), retention (building new knowledge from what has been observed) and reproduction (the application of the new knowledge though the actions of the apprentice). Including a stage to control the reproduction of the acquired knowledge, the reproduction stage, provides additional control over the apprentice's abilities after learning.

If motivation is added as an overall feature, the process described by Demiris and Hayes (Demiris & Hayes, 2002) becomes a reflection of the key aspects of learning by observation in social sciences (Bandura, 1977). The learning process can also be improved with an evaluation stage (see section 2.2.3) which allows the agent to measure how its performance is affected when acquiring knowledge and also when applying the acquired knowledge to execute a task (Wood, 2008; Billing *et al.*, 2010).

Despite the contributions of social learning theories, a computational approach to learning by observation requires the definition of a learning algorithm. The following section shows the different possibilities for the learning algorithm.

2.2.2 Possibilities for the Leaning Algorithm

This section provides an overview of the different possibilities for the learning algorithm. One of the most important features of a computational approach to learning by observation is defin-

ing how to build new knowledge from what is being observed (the demonstration data) and how this knowledge is used by the agent. The literature overview shows that different possibilities such as sequencing, classification learning, neural networks, case-based reasoning and inductive logic programming can be used for the learning algorithm.

The definition of the learning algorithm builds on the presumption that the demonstration data represents the action executed by the agent and the conditions holding for that action at the time the expert selected the action to execute (Argall *et al.*, 2009), as explained section 2.1.3. This way, the learning algorithm provides the agent with the ability to estimate actions for the current conditions.

Sequencing is one of the most common possibilities for the learning algorithm in the reviewed approaches (see section 2.2.5). It relies on the sequence in which the demonstration data was observed on the expert to provide the agent with a plan on how to achieve a task. This plan consists of a sequence of actions that has to be followed by the agent, so that it accomplishes a task. It provides the agent with a guideline on how to go from one state (the conditions currently faced by the agent), to a goal or sub-goal state (the ideal conditions holding when the task is completed) (Argall *et al.*, 2009)

The sequencing possibility is best suited for situations where behaviours are correlated and a chain of repeating events can easily be determined such as in repetitive tasks (Veeraraghavan & Veloso, 2008). It is also closely related with sequence learning in humans since it handles the same kind of problems, such as predicting the elements of a sequence based on the preceding element, finding the natural order of the elements in a sequence and selecting a sequence of actions to achieve a goal (Clegg *et al.*, 1998; Sun, 2001).

Earlier computational approaches to learning by observation such as String Parsing (SP) (Byrne, 1999) and Associated Sequence Learning (ASL) (Heyes & Ray, 2000) use the sequencing possibility for the learning algorithm. The main purpose of these theoretical approaches was to stimulate the development of learning by observation models and to serve as a guideline for theoretical experiments. They lack important features such as the references for the kinds of inputs that can be used for observation. There is also no clear indication of the kind of information that is used for observation (Alissandrakis *et al.*, 2002).

The most common approaches inspired in sequence learning use neural network models, such as recurrent back-propagation networks (Frasconi *et al.*, 1995; Giles *et al.*, 1995), to solve these problems. The Dynamical Recurrent Associative Memory Architecture (DRAMA), from

Billard and Hayes (Billard & Hayes, 1999), is also inspired in sequence learning and uses a recurrent neural network for its learning algorithm. Despite the difficulties in generalizing approaches on robotic systems (due to the differences in robot hardware), DRAMA has been successfully implemented in several robotic platforms and contexts.

The recurrent neural network in DRAMA allows learning of spatio-temporal regularities and time series in discrete sequences of inputs. It is capable of dealing with great amounts of noise on its inputs due to the neural network capabilities. The neural network also has low resource requirements for training. Experiments show that, even with basic hardware and very limited computational capabilities, robots were still able to carry out real time computations and on-line learning of relatively complex tasks (Billard & Hayes, 1999). However, this is only true when sensor input data is not complex (for example, numerical values). For more complex data like software objects (that are commonly used by software agent approaches), the number of inputs of the neural network increases, causing a decrease in its performance.

A simpler solution for sequencing is presented in several approaches to learning by observation (Byrne, 1999; Heyes & Ray, 2000; Cederborg *et al.*, 2010; Kulic *et al.*, 2011; Billing *et al.*, 2010, 2011; Fonooni *et al.*, 2012). These approaches reflect the solutions for the problems of sequence learning in the structures that are used to store the observed data. These structures preserve the temporal relations so that future reconstructions of the sequence on which the data was observed are possible.

Some of these approaches (Byrne, 1999; Heyes & Ray, 2000) use linear structures such as vectors and lists to store the data, which intrinsically preserves the temporal relations (for example, the next element in the sequence is the following element in the linear structure). The agent can easily recall these sequences using pattern recognition or other comparison methods, such as the nearest neighbour algorithm used in (Cederborg *et al.*, 2010), to find a specific element in the sequence and thus determine where to start following it. Important actions can also be easily identified by paying attention to their repetition on multiple observations, especially if these repetitions happen under different conditions (Byrne, 1999).

More recent approaches (Cederborg *et al.*, 2010; Kulic *et al.*, 2011) improved the method of storing the observed data using tree structures instead of linear structures. This allows the representation of different approaches for the same task and also the generalization of a task (for example, the initial movements for lowering the body using the legs are the same whether it is for picking an object in the ground or for kneeling). Parts of a sequence can be aggregated

in other sequences when both share the same transition of elements (for example, the initial movements for kneeling and lowering the legs could belong to the same sequence). The tree structure enables the agent to represent its knowledge as a decision tree, providing a choice for each element in the sequence (for example, after the initial movements the agent chooses if it wants to kneel, following one branch of the tree, or pick an object, following another branch of the tree).

The agent improves its ability to make choices in the sequence if the elements of the sequence contain the actions and the conditions holding before those actions were executed (the demonstration data). This allows the agent to compare the conditions faced when proposing actions with the conditions of each choice. This kind of decision making is similar to a decision tree, which is the strategy used by Sammut and his colleagues (Sammut *et al.*, 1992; Isaac & Sammut, 2003). The ability to make choices is also found in human sequence learning, as described by Sun (Sun, 2001).

A different way of preserving relations between the demonstration data acquired from the expert is proposed in the Predictive Sequence Learning (PSL) from Billing and his colleagues (Billing *et al.*, 2010, 2011) and Fonooni and his colleagues (Fonooni *et al.*, 2012). The predictive sequence learning uses fuzzy rules to describe the relations between the elements of a sequence as a hypothesis. The relation between an element and the next one in the sequence is described by a probability. For example, the higher the number of times element A is observed after B, the higher the probability of B being the one following A.

In addition to sequencing, other possibilities for the learning algorithm include generalizing the acquired information, making analogies between the acquired information and the new problems and categorizing the information and determining which set of categories the new problems belong (Argall *et al.*, 2009). This allows the agent to face future conditions that were not observed on the expert. Unlike the sequencing possibility, there is no specific sequence of actions to follow. The agent determines what actions it should perform supported exclusively by the conditions it is currently facing.

One way of generalizing the acquired information is through the Bayesian algorithms (Inamura *et al.*, 1999; Rao *et al.*, 2004; Sugimoto *et al.*, 2012). These algorithms provide a statistical approach, based on the Bayes theorem, to build a probabilistic model of what was observed. The probabilistic model is built with the demonstration data acquired (by observation) from the expert. According to Rao and his colleagues (Rao *et al.*, 2004), the probabilistic model of the

Bayes theorem is the best choice for representing the brain tasks behind learning by observation, such as combining different sensor data and combining the sensor data with prior knowledge.

Besides the Bayesian algorithms, other kinds of supervised learning algorithms can be used for this possibility for the learning algorithm. The demonstration data, acquired when observing the expert, is used to train those algorithms so they can estimate actions for the conditions faced by the agent (Argall *et al.*, 2009). Neural networks are an example of an algorithm that generalizes the information acquired by observation (Billard & Hayes, 1999). The classification algorithms used in (Sullivan, 2011) are an example of an algorithm that categorizes the information acquired by observation and determines to which set of categories the new problems belong.

Other possibilities for the learning algorithm rely on reinforcement and inverse reinforcement learning. However, these possibilities are not very common because of the way the rewards are calculated. Usually, the examples provided by the expert do not provide clear rewards (Argall *et al.*, 2009). Using inverse reinforcement can be helpful in situations where reinforcements are not clear because it obtains the rewards from the examples (Russell, 1998; Ramachandran & Amir, 2007; Argall *et al.*, 2009).

The ability to learn by observation within the context of reinforcement learning is usually viewed as *"a source of reliable information that can be used to accelerate the learning process"* (Chernova & Veloso, 2009). Inverse reinforcement learning is normally used in many software approaches for learning by observation (Chernova & Veloso, 2009; Babes-Vroman *et al.*, 2011; Neu & Szepesvari, 2007; Neu & Szepesvári, 2009; Judah *et al.*, 2011; Lopes *et al.*, 2009) because it relies on sequences of state-action pairs, that can be provided by a demonstrator, to build a reward function (Babes-Vroman *et al.*, 2011). However, the proposed approaches only focus on the learning algorithm and do not provide additional information on the mechanisms used for acquiring the state-action pairs or even if the state-action pairs reflect a direct observation of the expert while it is performing the task.

Inverse reinforcement learning approaches also have some limitations when compared with other possibilities because they provide multiple reward functions. Even though these reward functions can be optimal for performing the task, they do not represent the expert's reward function (Babes-Vroman *et al.*, 2011; Judah *et al.*, 2011; Neu & Szepesvari, 2007). This requires the algorithm to solve the problem multiple times so that it is possible to find out the reward functions that are closer to the expert, which might be computationally expensive when considering

large problems (Judah *et al.*, 2011).

Inverse reinforcement learning algorithms are also more sensitive to the quality of the data provided in the demonstrated examples. Like most of the supervised learning algorithms it is hard to provide proper solutions for state spaces that have never been observed (Neu & Szepesvari, 2007). However, in inverse reinforcement learning, the quality and the amount of examples acquired by the apprentice has direct influence in the way the reward function is calculated. If the examples do not completely specify the rules for performing the task, it becomes harder to obtain an exact reward function. This increases even more the number of reward functions proposed by the algorithm (Judah *et al.*, 2011). Therefore, many approaches require additional techniques to increase the quality of the observed demonstrations such as requesting the expert to provide feedback (Chernova & Veloso, 2009) or specific demonstrations (Judah *et al.*, 2011; Lopes *et al.*, 2009), or use probabilistic methods to improve the agent's knowledge (Neu & Szepesvári, 2009; Babes-Vroman *et al.*, 2011).

One of the major problems of using probabilistic methods to increase the agent's knowledge is that it increases the computational requirements of the algorithm. Furthermore, many probabilistic methods use statistical analysis of the surrounding environment to determine expert intentions (Babes-Vroman *et al.*, 2011). This requires the apprentice to collect additional information that does not come directly from the expert and makes the quality of learning dependent on the quality of the knowledge of the model of the environment (Neu & Szepesvari, 2007). One possible solution for that is provided by Neu and Szepesvari (Neu & Szepesvári, 2009). They use a set of tunable weights to optimize the way the reward function is calculated. This, however, continues to require an additional computational effort.

Another way of increasing the quality of the calculated reward function in inverse reinforcement learning is using expert feedback whenever the agent is uncertain of the action to perform. Chernova and Veloso (Chernova & Veloso, 2009) use automatically calculated thresholds to provide the agent with a measure of its uncertainty. The threshold decides when it is necessary to ask the expert for new data, even though it might be difficult to calculate (Judah *et al.*, 2011). Other problems that might come from expert feedback is making too many bad questions, that is, questions on what not to do (Judah *et al.*, 2011). Judah and his colleagues (Judah *et al.*, 2011) try to reduce this problem by incorporating a query selection mechanism in the apprentice. The mechanism filters the questions so that the apprentice only makes the ones that produce new knowledge.

Lopes and his colleagues (Lopes *et al.*, 2009) present a different method of filtering the apprentice questions that reduces the amount of examples required to build the reward function. The filter uses a set of known probabilities on state transition to compute the states that are most informative. This, however, requires additional knowledge on the probabilities of one state transitioning to another, which might not always be available. Despite being able to increase the quality of the calculated reward function without requiring too much computational effort, expert feedback solutions get outside the scope of a pure leaning by observation approach, where the expert agent does not participate in the learning process and might even be unaware that it is being observed.

Neu and Szepesvari (Neu & Szepesvari, 2007) propose a mixed solution that includes both inverse reinforcement and other supervised learning algorithms. They use the supervised learning algorithms to penalize the deviations of the apprentice and the inverse reinforcement learning to fine tune a reward function that provides the agent with the ability to perform a task. This solution, however, is more complex to implement and does not solve the problem of generating more than one optimal reward function.

Inductive logic programming (ILP) is another possibility for the learning algorithm, proposed by Konic and Laird (Könik & Laird, 2006). It extracts first order rules from structured data obtained through observation. However, this structured data does not take the form of the demonstration data used in all other strategies, the condition-action pairs. Also, one of the particularities of inductive logic programming is the focus on reflecting the expert's rules on the data provided for observation. This forces the apprentice to learn not only the conditions holding when the actions were selected for execution but also the reasoning behind the expert's decisions, which requires more information to be provided for observation (Könik & Laird, 2006).

To allow agents to discover the expert's internal reasoning, the process of observing an expert requires several complex knowledge sources. These sources can include behaviour traces, expert goal annotations and background knowledge. The knowledge acquired when observing these sources is partitioned into a hierarchy and represented as first order rules. To store this knowledge, Konik proposes an episodic database. This database is an extension of Prolog that improves the temporal relations between the observed data (Könik & Laird, 2006).

However, the kind of complex data required by the inductive logic programming possibility escapes the concept of learning by observation, on which the observed data should only repre-

sent features that are visible, like the state of the environment, the agent visual attributes and the actions (see section 2.1). For this reason, inductive logic programming is not a good candidate for an approach to learning by observation where agents observe the actions (and the conditions holding for those actions) performed by an expert while executing a task.

In addition to the learning algorithm, an approach to learning by observation must also provide solutions for all features of the learning process. The way the agent evaluates the information acquired by observation is one of these features. The following section shows several approaches on how to evaluate the agent's knowledge.

2.2.3 Evaluating the Agent's Knowledge

This section explains the importance of evaluation and describes several approaches to evaluate the agent's knowledge.

Including an evaluation stage in the learning process provides the agent with a simple motivation mechanism. The agent can be intrinsically motivated to learn because it knows when it has acquired sufficient knowledge to perform a task on its own or because it detects that portions of its knowledge need improvement and thus require the agent to go back learning (Wood, 2008; Billing *et al.*, 2010). The ability to enhance the agent's knowledge through new observations is an important factor for learning by observation because one of the downsides of learning by observation is the fact that the agent's knowledge is limited to what it was able to observe (Argall *et al.*, 2009).

An appropriate evaluation of the agent's knowledge allows the agent to measure how its performance is affected both when it is consolidating the information acquired by observation or when executing actions (Wood, 2008; Hajimirsadeghi & Ahmadabadi, 2010). The agent's performance can be captured by a confidence factor that represents the agent's confidence on its knowledge. The value of the confidence factor determines if the agent is prepared, or not, for selecting actions for the current conditions (Billing *et al.*, 2010).

One way of determining the confidence factor is using the information provided by the learning algorithm (Chernova, 2009). When proposing actions, the learning algorithm used on Chernova's approach (Chernova, 2009) also provides a measure of its confidence on the proposed actions. However, this confidence only reflects the distance between the agent's knowledge and the new problems. It is not possible to show how the agent evolved and how many times in the past it has been able to propose the correct actions.

Evaluating the actions executed by the agent and including the results of this evaluation on the confidence factor is essential for determining what portions of the agent's knowledge need improvement (Wood, 2008). An evaluation that operates when the agent is learning and when it is using the acquired knowledge to select actions to execute a task, allows agents to know when it is necessary to observe experts and when they are ready to execute actions. One way of evaluating the agent while it is executing actions is through external feedback, which can be provided by the expert agents. Several authors (Sullivan, 2011; Hajimirsadeghi & Ahmadabadi, 2010; Chernova, 2009) use specialized experts, or teachers, to monitor and reinforce the agent's actions. The teachers measure the agent's performance and provide the necessary feedback for evaluation.

In addition to monitoring and providing basic feedback, the relation between the teacher and the agent can be extended by enhancing the communication protocol between them. This way, teachers can take corrective measures like providing the agent with the appropriate action for the current conditions whenever the agent tries to execute an incorrect action (Hajimirsadeghi & Ahmadabadi, 2010). The teacher can also override the agent's decision of whether executing actions to perform a task or observing experts to acquire more knowledge (Sullivan, 2011). The extended communication also allows the agent to request the teacher to perform a specific task (Chernova, 2009).

However, enhancing the communication protocol requires an additional effort in designing the expert agents since they need to know how to communicate with the agents besides knowing how to perform the requested tasks. This also requires apprentice agents to wait for the teacher to be available to communicate with them, which can extend the amount of time spent learning. This does not happen when the expert plays a passive role in observation. In addition, using teachers approximates the approaches to learning by teaching, which is something beyond learning by observation.

Therefore, an approach to learning by observation should focus on solutions where the expert has a minimal role. The following section shows the importance of mimicking the behaviour of mirror neurons for the evaluation of the agent's knowledge.

2.2.4 The Mirror Neurons

This section shows how mimicking the behaviour of mirror neurons is important for learning by observation and how it can be emulated in a computational approach. The section also

shows how mimicking the behaviour of mirror neurons can be useful for evaluating the agent's knowledge.

Under a neurological perspective, learning by observation in humans and superior mammals is defined by the mirror system, which includes the canonical and the mirror neurons. Experiments on macaque monkeys show that these neurons are involved in the tight coupling between what is being observed and the motor control. Experiments also show that mirror neurons are active when the monkeys observe and also when they make a specific movement. This allows observers to ”feel” like they are performing the actions they observe (Rizzolatti *et al.*, 1996, 2000; Ramachandran, 2000; Demiris & Hayes, 2002).

The discovery of the mirror system was significant for understanding learning by observation since it appears to provide the direct transformations from the observation of an action to its execution. It is postulated that the mirror neurons are the basis for learning by observation and other complex mechanisms like understanding actions and social interaction (Maistros & Hayes, 2001).

Neuro-imaging studies suggest that the human mirror system is sensitive to sensory-motor experience. It was verified by experimentation that when ballet experts observe ballet sequences performed by someone from the same gender they present stronger responses from the mirror neurons, than when observing similar sequences performed by the opposite gender. This sensitivity is a strong indication that the properties of mirror neurons are acquired through learning (Calvomerino *et al.*, 2006).

The activation of mirror neurons, whether apprentices are observing or performing the actions, suggests that learning by observation allows apprentices to place themselves in the position of the experts. For Demiris and Hayes (Demiris & Hayes, 2002) it is as if the apprentices were able to generate several alternatives for the actions they observe on another expert, that is, as if apprentices were proposing actions for the conditions faced by the observed expert. This allows apprentices to understand the observed behaviour, thus improving the quality of the acquired knowledge. It also allows apprentices to measure the confidence on their knowledge by determining if they are able, or not, of proposing the same actions as the expert (see section 2.2.3).

Several computational approaches to learning by observation attempted to mimic the behaviour of the mirror neurons and especially their ability to inhibit the motor controls, which allowed the agent to propose actions without effectively executing them. The most common strat-

egy to achieve this behaviour is though forward models. They are used in several approaches (Demiris & Hayes, 2002; Maistros & Hayes, 2004; Rao *et al.*, 2004; Lopes & Santos-Victor, 2007; Sugimoto *et al.*, 2012) as a way of building an internal representation of the world for the agent.

Through the internal representation of the world, the agents are able to predict how the actions they propose affect the state of the world without actually executing the actions (Demiris & Hayes, 2002; Rao *et al.*, 2004; Lopes & Santos-Victor, 2007). This is a safe way for the agent to simulate the expert's internal reasoning while observing the actions it executes because the actions proposed by the agent have no effects in the environment. This effect is similar to the one produced by the mirror neurons.

Some of the implementations of forward models (Demiris & Hayes, 2002; Maistros & Hayes, 2004; Lopes & Santos-Victor, 2007) are specifically designed for robot agents and usually reflect a physical device that inhibits the motor activity. However, Rao and his colleagues (Rao *et al.*, 2004) show that it is also possible to implement these forward models using Bayesian algorithms. Recent approaches (Cederborg *et al.*, 2010; Hajimirsadeghi & Ahmadabadi, 2010; Kulic *et al.*, 2011) show that Hidden Markov models can also be used to mimic the behaviour of mirror neurons.

Like the Bayesian algorithms, the Markov models create an abstract model that associates the effects with the actions that caused those effects. The models help the agents understand the new actions, providing them with an abstract representation for them. The abstract representations represent both the observed actions (executed by the experts) and the agent actions (proposed and executed by the agent) (Cederborg *et al.*, 2010; Hajimirsadeghi & Ahmadabadi, 2010; Kulic *et al.*, 2011).

Heyes and Ray (Heyes & Ray, 2000) also use abstract representations of the agent actions. They state that through abstract representations it is possible to achieve an effect that is similar to the forward models and hence to the mirror neurons. The abstract representations provide a clear distinction between the representation of an action and its execution. This allows the agents to handle their actions without necessarily executing them (Heyes & Ray, 2000).

If all agent actions are abstract representations, the same mechanisms can be used to propose actions both when the agent is observing and when it is executing a task because the proposed actions are not immediately executed and can even be discarded. The agent is able to ”feel” like it is performing the actions it observes if it proposes actions for the conditions holding for

the actions it observes because the agent is using its mechanisms to propose actions for the conditions faced by the expert.

2.2.5 Approaches to Learning by Observation

This section provides an overview of the existing approaches to learning by observation. Despite significant advances on computational approaches to learning by observation there is still a lack of an established structure on which to organize them (Argall *et al.*, 2009; Tan, 2012). The analysis of the most recent approaches on learning by observation reveals that the ability to directly observe the agent and its activities is still exclusive to robotics and that a global model for this kind of learning is still missing (Hajimirsadeghi & Ahmadabadi, 2010; Cederborg *et al.*, 2010; Sullivan, 2011; Billing *et al.*, 2011; Kulic *et al.*, 2011; Fonooni *et al.*, 2012; Sugimoto *et al.*, 2012; Tan, 2012). Only with this global model it is possible to determine the important components of a learning architecture and how these components interact with each other.

The major problem in most of the surveyed approaches to learning by observation is the focus on a specific problem, which is usually solved by defining a learning algorithm (see section 2.2.2). Almost all approaches lack an overall view of the learning process which should comprise the methods for finding and observing an expert, for storing and interpreting the observed data (the learning algorithm) and for applying and evaluating the acquired knowledge (Tan, 2012).

With the exception of Machado's approach (Machado, 2006; Machado & Botelho, 2006), all other approaches concerning the direct observation of the agent and its activities are related to robotic agents. Software approaches for learning by observation are usually focused on the learning algorithm and don't provide details on how the training data is obtained or even if it reflects state information or agent activities. Nonetheless, inverse reinforcement learning approaches like (Chernova & Veloso, 2009) can provide useful contributions for the learning algorithm such as insights on how to automatically calculate uncertainty thresholds (Chernova & Veloso, 2009) and ways of improving the acquired knowledge with expert feedback (Chernova & Veloso, 2009; Judah *et al.*, 2011; Lopes *et al.*, 2009).

Even Machado's work has limitations since it only addresses the problem of learning vocabulary and it does not allow generalizations of the acquired knowledge. This means that her agents are not able to learn control knowledge neither are capable of dealing with conditions that are different from those observed on the expert.

From the surveyed approaches, Sullivan's approach (Sullivan, 2011) is the one who combines the most interesting contributions for the approach described in this thesis. His approach describes a learning process consisting on the observation of an expert, learning from the demonstration data, applying the acquired knowledge in new problems and expert evaluation (teacher appraisal) of the executed actions. His approach is one of the few that uses the analogy possibility for the learning algorithm (see section 2.2.2). He uses classification algorithms, which makes his agents capable of dealing with conditions that are different from those observed on the expert.

In addition, the experts actively participate in the evaluation of the agent actions. The agents can return to a learning stage (where they exclusively observe experts) whenever new knowledge needs to be acquired. The ability to return to a learning stage is rarely seen in most of the learning by observation approaches. However, the decision to return to learning is not made by the apprentice, it is the expert that forces this change (on the apprentice) whenever it detects that the apprentice is not correct (Sullivan, 2011).

The following section presents the conclusions of the survey on embodiment and on learning by observation and some critical comments on the reviewed approaches.

2.3 Critical Comments and Conclusions

The literature overview shows that computational approaches to learning by observation are advantageous for machine learning. However, with the exception of Machado's approach (Machado, 2006; Machado & Botelho, 2006) and some contributions from robotic approaches, the majority of software approaches for learning by observation do not take into account the direct observation of the software agent and of its activities. In these cases, the preference for robotic approaches is natural because learning by observation is an interaction between tangible entities.

The software approaches that are closer to learning by observation are limited to observing the effects of actions. However, several authors (Bandura, 1977; Byrne, 1999; Mataric, 1997; Botelho & Figueiredo, 2004; Machado, 2006) defend that learning by observation would become a laborious process if only the effects of actions were taken into account, especially when the agent does not know the effects of its actions *a priori*. Learning by observing only the effects of the actions is also impossible when those actions do not change the state of the observable

world. Observing the actions and body movements is advantageous for learning by observation because it simplifies the learning process and increases the learning speed.

For learning by observation of an expert's actions to happen in the software world, software agents need some kind of representation for their constituents and for the actions they perform - the software image. With the exception of Machado's approach (Machado, 2006; Machado & Botelho, 2006), the reviewed literature does not provide any direct solution for this visual representation (see section 2.1). However, it provides insight on how to collect the information required for the visual representation (see section 2.1.2) and on the features required for the visual representation (see section 2.1.3 and 2.1.4).

Machado's proposal (Machado, 2006; Machado & Botelho, 2006) provides a good starting point for addressing the problem of creating a software image. However her approach lacks important features such as describing the kind of inputs the agent can collect from the software environment (the sensors), the conditions holding when the agent decided to execute an action and a historical record of the actions executed by the agent and the conditions holding for those actions (see section 2.1.6).

In addition to the software image, a computational approach to learning by observation must also define how the information, obtained from observation, is stored and how it is used - the learning algorithm. The survey on learning by observation shows that there are several possibilities for the learning algorithm (see section 2.2.2). The sequencing and the classification possibilities are the ones that provide the best advantages and the least complexity for an approach to learning by observation.

The classification possibility allows the agent to extend its knowledge to conditions that have not been observed and the sequencing possibility allows the agent to easily learn sequences of actions with different alternatives. The sequencing possibility also defines a way of storing the agent's knowledge using tree structures (Cederborg *et al.*, 2010; Kulic *et al.*, 2011). One of the advantages of using tree structures is the ability to represent different approaches to the same task. Given the advantages of these two possibilities, the proposed approach will combine both of them.

The survey on learning by observation also shows that one of the major problems of computational approaches is the focus on the learning algorithm and the lack of a global model for learning by observation (see section 2.2.5). A general approach to learning by observation cannot depend on a specific learning algorithm because it may hinder the ability to adapt to

different circumstances. The approaches for learning by observation usually reflect specific problems, which imply adaptations of their contributions whenever they are applied in new domains. Even the most recent approaches (Hajimirsadeghi & Ahmadabadi, 2010; Cederborg *et al.*, 2010; Sullivan, 2011; Billing *et al.*, 2011; Kulic *et al.*, 2011; Fonooni *et al.*, 2012; Sugimoto *et al.*, 2012) follow this trend.

As section 2.2.1 shows, social learning theories provide a good ground for describing the learning process. They also provide insight on determining the experts from which it is potentially possible to learn, those whose structures and abilities are identical to the agent that wants to observe them.

In normal circumstances, although the expert and the agent have the same features it does not necessarily mean that the expert is performing the actions that are necessary for the apprentice to learn how to perform a specific task. However, on all the surveyed approaches this aspect is disregarded and the experts only execute the actions that are necessary to perform the task to learn. Even though this is an important subject, the problem will be left outside of the scope of this thesis due to the lack of time. The same simplifications will be applied when testing the approach (see chapter 5).

To the best of our knowledge, the only computational approach that defines a learning process that follows the social learning theories is described by Demiris and Hayes (Demiris & Hayes, 2002) (see section 2.2.5). However, the process described on their approach misses an important aspect, the evaluation of the acquired knowledge (Wood, 2008; Billing *et al.*, 2010). The evaluation provides the agent with a measure of the confidence of its knowledge. It is also a way of providing the agent with a simple motivation to learn (when it stops being confident on its knowledge) or to stop learning (when it is confident on its knowledge), as described in section 2.2.3.

Despite the possibilities for determining the confidence factor presented in section 2.2.3, a simpler way of doing it is testing the agent's knowledge while it is observing the experts (Billing *et al.*, 2011). A neurological perspective of learning by observation provides an important contribution to the evaluation of the agent's knowledge - the behaviour of mirror neurons (see section 2.2.4). Mimicking this behaviour allows agents to use the same mechanisms for proposing actions when observing and when preparing to execute them. Thereby, an agent can determine if it is capable of taking the same decisions as the observed expert by proposing actions for the conditions holding for the actions it observes.

This allows the agent to experience the observed actions as if it was actually executing them (Ramachandran, 2000). The agent can therefore propose actions for the conditions holding before the observed actions were executed and compare the proposed actions with the observed actions. If the proposed actions are the same as the ones observed on the expert, the agent is prepared to face that specific situation.

The survey shows several approaches for mimicking the behaviour of mirror neurons (Demiris & Hayes, 2002; Maistros & Hayes, 2004; Lopes & Santos-Victor, 2007; Rao *et al.*, 2004), most of them exclusive to robotics. The simplest solution is abstracting the agent actions, that it, explicitly separating the representation of an action from its execution (Heyes & Ray, 2000; Kulic *et al.*, 2011). This separation provides the agent with control over the execution of the actions it proposes.

The ability to receive external feedback from specialized experts is also an important contribution to evaluation (Hajimirsadeghi & Ahmadabadi, 2010; Sullivan, 2011). It allows the agent to know if the actions it executes are appropriate or not, which complements the way the agent updates the confidence on its knowledge. However the ability to receive feedbacks from specialized experts is closer to learning by teaching than learning by observation. In learning by observation the expert agent has a passive role in the learning process.

In conclusion, the survey on learning by observation shows that the majority of the computational approaches depend of a specific domain and almost all of them cannot be directly applied to software agents without substantial changes. With the exception of Machado's approach (Machado, 2006; Machado & Botelho, 2006), all other software approaches ignore the possibility of directly observing the software agent and its activities and limit themselves to observing state changes. Therefore, learning by observation in software environments needs a different approach where the interactions between software elements must also be taken into account.

Chapter 3

The Software Image

This chapter describes the proposed approach for the ”visible” representation of software agents - the software image. It shows the elements that make up the software image and the additional functionalities that the proposed approach introduced to previous work regarding the software image (Machado, 2006; Machado & Botelho, 2006). The chapter also describes the software image meta-ontology and the software created for the software image. An overview of the proposed software image is presented in (Costa & Botelho, 2011).

The ability to see other agents is essential for learning by directly observing the actions of an expert while it is performing a task. However, software agents are unable to see themselves or the others in the same way tangible entities such as robots see themselves and the others (see section 2.1). Therefore, the approach for learning by observation proposes a software image that allows software agents to observe themselves and other agents. The proposed software image is an accessible representation of the agent that is used by any agent that wants to observe the represented agent.

Although it was designed for the purposes of this research, which is learning control knowledge by observing the actions executed by other agents, the proposed software image can also be used in other circumstances. For example, it could be used for learning vocabulary as in Machado's work (Machado, 2006; Machado & Botelho, 2006) or, in general, it may contribute to embodiment. Additional work will incrementally reveal the characteristics of a software image that is independent of any particular use. Despite this more general long term goal, this chapter describes the current version of the software image, which was created specifically for learning by observation.

Although the visual information is an important source of information for several species

of animals, including humans and superior mammals, it is still hard for computers to extract the same amount of information from vision sensors as we do. Nonetheless, looking at nature, it is possible to identify other ways of acquiring information that is as useful as the visual information. Examples of this can be seen in bats with their ultrasonic senses and in dogs with their smell senses. Nature shows us that it is possible to adapt sensors to the type of environment where the organism lives.

Using the same concept in the software world, it is possible to develop rich information sensors that are specialized in providing information about software agents. Therefore, the act of observing a software agent does not imply the use of computer vision; instead the agents use specialized sensors that read meta-data about the observed agent. This meta-data is what we call the agent's "visible" software image or simply the software image. Figure 3.1 shows a representation of the software image and its relation with the software agent.

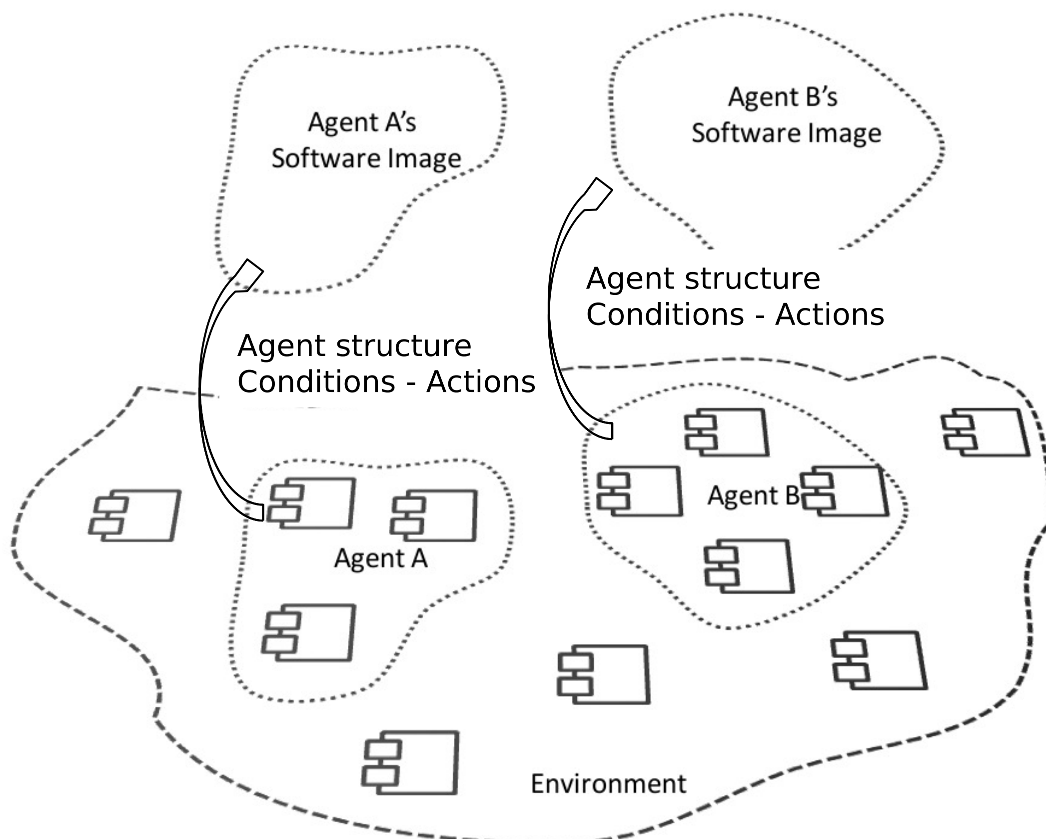


Figure 3.1: The software image in relation with the software agent

The software image represents software objects and the relationships among them. As figure 3.1 shows, the software image provides relevant information on important features of the software agent such as its constituents, the actions it executes and the conditions holding when it decided to execute those actions. The software image is universally accessible and has a domain

independent description so that it can be observed and understood by other agents (see section 3.2). An agent can only have one visible appearance and an instance of a software image can only represent one agent.

Following Machado's work (Machado, 2006; Machado & Botelho, 2006), the proposed software image is arranged in two categories - the static image and the dynamic image (see section 3.2). The static image is immutable (does not change over time) and describes the constituents and capabilities of the agent. Software agents are regarded as collections of parts with sensors, visible attributes and actuators (see section 3.2.1). The dynamic image changes with time and describes the current and past actions executed by the agent along with the conditions holding when the agent decided to execute those actions (see section 3.2.2).

The proposed software image improves Machado's proposal (Machado, 2006; Machado & Botelho, 2006) by including the agent sensors in the description of its components (see section 3.2.1), combining the information on the state of the software environment (provided by the agent sensors) with information on important aspects of the agent's internal state (the visible attributes) with the observed actions and enabling the representation of composite actions, that is, actions composed of sequences of simpler actions (see section 3.2.2).

Other important improvements include the ability to store historic data on the agent actions and on the conditions holding for those actions (see section 3.2.2), and the use of an ontology to represent the agent sensors, actions and visible attributes, the tasks to be accomplished and the relationships that exist between those elements (see section 3.3).

The proposed software image also improves its computational realization and defines a new protocol for observation (see section 3.4). The software developed for the software image allows agents to build their own software image, update it and compare it with the software images of other agents. The software requires only a minimal intervention in the original agent source code to provide software agents with a software image.

The following section shows the importance of two major features of the proposed software image: The inclusion of the agent sensors in the description of the agent's components and capabilities and the inclusion of the conditions holding for the executed actions in the data provided for observation.

3.1 The Importance of the Agent Sensors and Perception

This section explains why it is important to describe the agent's sensing capabilities in the software image, specifically in the static image. It also presents two possibilities for the conditions holding for the executed actions. One possibility is using the perspective of the agent that is executing the actions. The other possibility is using the perspective of the agent that observes them.

When considering software environments, there is a common notion that the information acquired from them is singular and universally accessible. All the components of a program are able to obtain the same kind of information from the environment. Under this perspective, describing the sensing abilities of software agents makes no sense because the state of the environment is the same for all agents.

However a closer examination shows that the components of a program may only have access to the elements of the software environment that are important to them. Software agents can have sensors that constrain the kind of information they receive from the environment. The ability to sense distinguishes the agents from the remaining elements of the software environment because it constrains them to express the world the way they sense it (see section 2.1.1). The proposed software image includes the sensors in the representation of the agent constituents because they represent the way the agent understands the world.

The agent sensors are also an important aspect for learning by observation because it is more likely for the observer to understand the reasoning of the expert agent if both share the same understanding of the world (Demiris & Hayes, 2002; Ramachandran, 2000). For example, an agent with a sensor that is capable of perceiving an array of colours (the colour agent) acquires more knowledge by observing agents with the same kind of sensor than by observing agents with a colour blind sensor (the greyscale agent), which is only capable to perceive in greyscale. The colour agent would be missing important information provided by the colours that the greyscale agent cannot perceive. Thus, it is more efficient for the colour agent to learn from other colour agents than from a greyscale agent.

Another example is when one agent only has access to the color property of the objects whereas another agent only has access to the shape property of the objects. Even though the two agents perceive the same object they have different perspectives of that object. In this case, the two agents would never be able to learn by observing each other because they have different perspectives of the surrounding environment.

In general, when agents select actions for execution, their decision is supported by the information they hold about the world and by their current internal state. This information can be regarded as the conditions holding when the agent decided to execute those actions. The conditions contain information on the state of the environment, as it was acquired by the agent sensors at a given moment, and on aspects of the internal state that were important to decide the actions to execute (which are called the visible attributes, as explained in section 3.2.1).

Unlike the previous proposal (Machado, 2006; Machado & Botelho, 2006) where agents could only observe the action being currently performed, in the proposed software image, agents acquire snapshots of the current and past activity of the observed agent (see section 3.2.2). Each snapshot contains information on the actions executed in the time it takes for the agent to update its perception of the world along with the conditions holding at the moment the agent decided to execute those actions. The observed information can be seen as a condition-action pair, similar to those in the training sequences used for training machine learning algorithms, which is important for the learning algorithm as described in section 4.5.

The conditions holding for the observed actions play an important role in the information provided for observation. They may hold the perspective of the agent that observes the actions (the observer) or the perspective of the agent that executes them (the observed). Each of these possibilities has advantages and disadvantages. Using the perspective of the observer is advantageous because it does not require the observer and the observed agent to have the same representation of the world. The observer only needs to understand the actions it observes. The conditions are acquired through the observer's sensors and therefore it does not require the observer to have the same sensors as the observed agent. There is also no need to convert the conditions to the observer's own internal representation.

One of the disadvantages of using the observer's perspective in the conditions is the synchronization required between the observer and the observed agent. The observer needs to know exactly when to acquire the conditions for the actions it observes. The complexity of the synchronization increases in multi-agent environments because the conditions concerning the state of the environment can be changed by external factors, such as the activities of other agents.

Without this synchronization it becomes harder to guarantee which state of the environment before the actions were observed reflects the conditions holding when the agent decided to execute those actions. Figure 3.2 shows an example of how hard can it be for the observer to

choose the correct state of the environment for the conditions holding for the observed action.

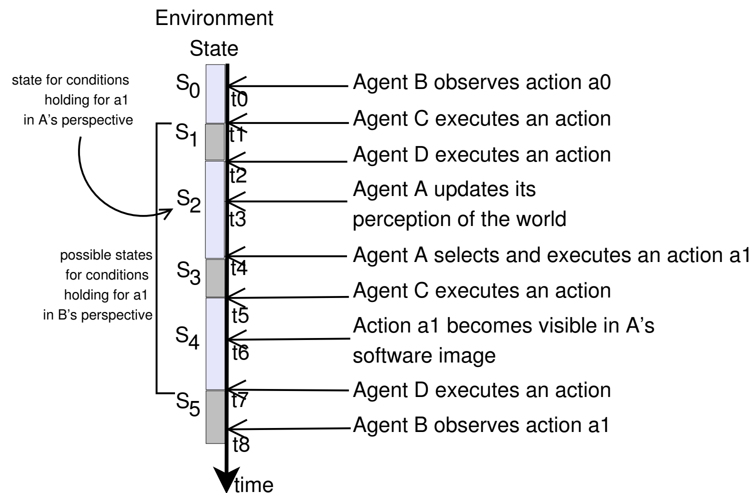


Figure 3.2: How the activities of other agents can hinder the ability to determine the conditions

The example in figure 3.2 shows that in the time it takes between observing *a0* and *a1* (from *t0* to *t8*) the state of the environment changes five times (from S1 to S5). Therefore, agent B has no way of knowing which of these states expresses the conditions holding when *a1* was selected and executed (S2) because it does not know exactly when agent A selected and executed *a1*. Without synchronization, agent B is only able to know when it has observed *a1* (at time *t8*), which is not the time at which A selected and executed the action (*t4*).

Another disadvantage of using the observer's perspective is when the observer has no direct access to the environment of the observed agent, such as when agents are running in distinct processes and have no knowledge on how to access the environment on the other process. For example, agent A, running on process P_a, is only able to acquire the state of the environment S_a, while agent B, running on process P_b, is only able to acquire the state of the environment $S_b(S_a \neq S_b)$. If agent A knows agent B and wants to observe it, it can read its dynamic image, but since it has no access to S_b, it cannot acquire the state of the environment of B. In such cases, even though agents are still able to exchange messages with each other, they would not be able to acquire the conditions holding for the executed actions without using specific mechanisms to get access to all the components of the software environment running on the process of the observed agent.

Using the perspective of the agent that executes the actions is advantageous because it does not require the observer and observed agent to be synchronized. It also allows observers to get access to information on the observed agent's past actions and on the conditions holding before those actions were executed - the history record (see section 3.2.2). The history record can hold

this information even when no agents are observing.

The major disadvantage of using the perspective of the agent that executes the actions is that it requires the observer to understand the world in the same way as the expert it observes, or at least, to understand the information in the conditions. In such circumstances, the agent can only observe experts as long as they have the same sensors and visible attributes, as explained in section 4.3.1, which is another reason to include the agent sensors in the agent software image.

Given the advantages and disadvantages of both possibilities, in the proposed approach, the conditions hold the perspective of the agent that executes the actions mainly because this is essential for the history record (see section 3.2.2). Without this perspective it would be impossible for observers to know the conditions holding in the past if they were not observing the agent at that moment.

Even though the proposed approach uses the conditions holding the perspective of the observed agent, other approaches may discard those conditions and use only the actions from the snapshots. As future work, the approach can be improved with methods of acquiring the conditions holding the perspective of the observer agent.

The following section presents the domain representation of the software image and the improvements concerning previous versions of it (Machado, 2006; Machado & Botelho, 2006).

3.2 The Elements of the Software Image

This section describes the elements that compose the software image. It shows the way these elements describe the agent's structure and its activities. It also describes the improvements on previous proposals for the software image (Machado, 2006; Machado & Botelho, 2006).

The software image comprises the static image and the dynamic image. The static image shows the structural relations between the agent's components and all the sensors and actuators considered as potential capabilities. The static image also shows the relevant aspects of the internal state which are manifested as visible attributes.

The dynamic image shows the current and past actions that were executed by the agent along with the conditions holding when those actions were selected for execution. The conditions comprise the state of the environment, as it is acquired by the agent sensors, and the instances of the agent's visible attributes at a given moment. Figure 3.3 shows the UML diagram of the software image.

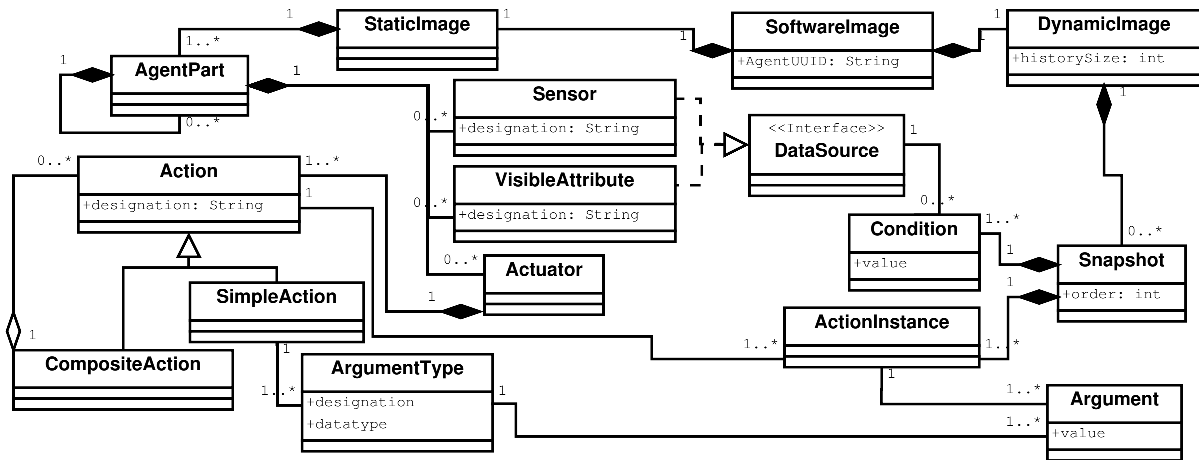


Figure 3.3: UML diagram of the software image

Figure 3.3 shows that the software image is composed of a static image and a dynamic image. The static image is a collection of agent parts, where each agent part represents a capability of the agent. A capability, or agent part, can have sensors that collect information from the environment, actuators that allow the agent to perform actions and visible attributes to show relevant aspects of the internal state. The agent parts can also be composed of other parts.

The dynamic image is composed of a sequence of snapshots of the agent activity. Each snapshot contains the instances of the actions executed by the agent, as they were invoked in that particular situation, and the conditions holding for those actions. The conditions consist of the information collected from the agent sensors and the instances of the visible attributes at a given moment. The sensors and visible attributes are regarded as data-sources because they are the sources of information for the conditions.

Figure 3.3 also shows that software images have a unique identifier (AGENTUUID), which is imprinted at the moment the software image is created (see section 3.4.1). This unique identifier allows an immediate distinction between identical software images, that is, software images whose static images share the same structure and the same elements. The identifier also provides the software agents with an easy way of recognizing their own software image. The following sections describe the static image and the dynamic image in detail.

3.2.1 The Static Image

The static image represents the agent components and its potential capabilities. It describes the agent as a composition of problem solving components called agent parts, each of which can have sensors, visible attributes and actuators (see figure 3.3). In Machado's version of

the software image (Machado, 2006; Machado & Botelho, 2006), the static image described software agents as a collection of parts with visible attributes and actuators, which in their turn provided the agent actions. However, as explained in section 3.1, agents must be characterized not only by what they can do (their actions) but also by what they can acquire, or perceive, from the environment (their sensors).

The sensors represent the agent mechanisms that are responsible for collecting information about the state of the world. They are essential for determining the way the agent understands the world (see section 3.1). The visible attributes represent important aspects of the internal state that influence the agent decisions and therefore need to be visible to others. The actuators represent the mechanisms that provide agents with the ability to change the state of the world and its internal state. They determine what the agent can do through the actions they provide. The actions are abstractions of the agent abilities. Depending on their complexity, the agent abilities can be simple actions or sequences of actions (a composite action).

The proposed software image allows observers to handle both the simple and the composite actions in the same way, which simplifies the observation of complex abilities. Instead of representing the ability with a single action that is harder to represent, the ability is decomposed in a sequence of simpler actions which are easier to represent. In addition, the simpler actions can be reused on other composite actions. The proposed software image also offers the possibility of instantiating the simple actions. The ability to instantiate an action enables it to be used in several different circumstances. The possible arguments of an action can be set up using the *ArgumentType* class (see figure 3.3).

The static image organizes the agent components in a tree. The agent parts and the actuators are sub-trees of the static image. The sensors, actions and visible attributes are leafs, or atomic elements, of the software image. This kind of description is advantageous for comparing the software images of two agents, as described in section 3.4.3. Static images can be distinguished from each other by their structure (the number of parts, sensors, visible attributes, actuators and actions) and by the kind of sensors, visible attributes and actions. Each kind of atomic element (sensor, visible attribute or action) has a distinct *designation*, which is described in an ontology (see section 3.3). Figure 3.4 shows an example of a static image of a software agent that sorts elements in lists.

The static image presented in figure 3.4 shows the relevant features and components of the list sorting agent. It describes the agent as being composed of two parts (AP1 and AP2). The

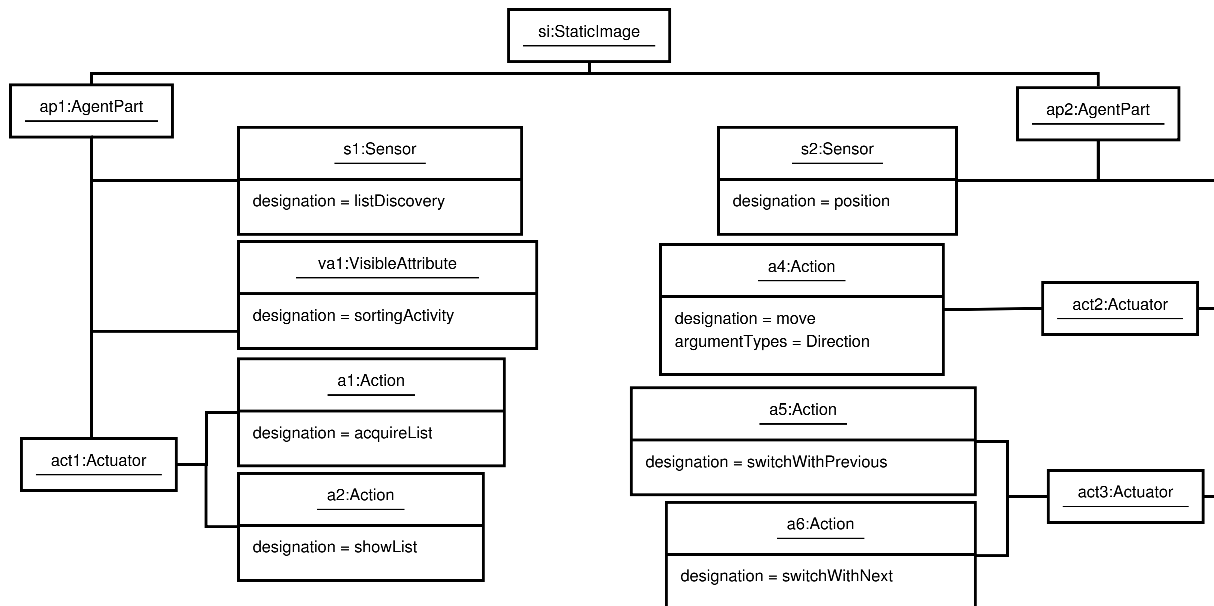


Figure 3.4: The object diagram of an example static image

first part of the agent (AP1) acquires lists from the environment using the *acquireList* action (A1). The part also displays lists with the *showList* action (A2). The agent part discovers the lists in the world using the *listDiscovery* sensor (S1) and displays information about its activity through the *sortingActivity* visible attribute (VA1).

The second part of the agent (AP2) sorts the acquired lists by switching an element in the list with the element in the previous position, using the *switchWithPrevious* action (A5), or with the element in the next position, using the *switchWithNext* action (A6). The agent part uses the *move* action (A4) to move itself along the list. The action has to be instantiated with a *Direction* for the agent part to know on which direction to move. The agent part knows its current position in the list using the *position* sensor (S2).

In addition to the innovations in static image, the proposed software image also improves the dynamic image, as described in the following section.

3.2.2 The Dynamic Image

The dynamic image holds information on the current and past actions that were executed by the agent, along with the conditions holding before the actions were executed. Unlike Machado's proposal (Machado, 2006; Machado & Botelho, 2006) where agents could only observe the action being currently performed, in the proposed software image, agents acquire snapshots of the activity of the observed agent.

The snapshot provides a global view of the actions performed by the agent in the time it takes to update its perception of the world (see section 3.4.2). Therefore, it is possible for a snapshot to hold more than one action, especially when the agent has several parts and each of them is able to execute actions in the time it takes to update the perception. Given that actions can be instantiated, as explained in section 3.2.1, the snapshot holds instances of the executed actions as they were invoked in a particular situation. The *Argument* class holds information on the arguments used to invoke those actions (see figure 3.3).

In addition to the executed actions, the snapshot also shows the conditions holding before those actions were executed. As section 3.1 explains, the conditions in the snapshot hold the agent's perspective. They represent the important aspects of the agent's internal state (the visible attributes) and the current understanding of the state of the world (the information acquired by the sensors) at the time the agent selected the actions for execution because this information plays an important role in deciding which actions to execute. Figure 3.5 shows an example of a snapshot that can be observed on the agent whose static image is displayed in figure 3.4.

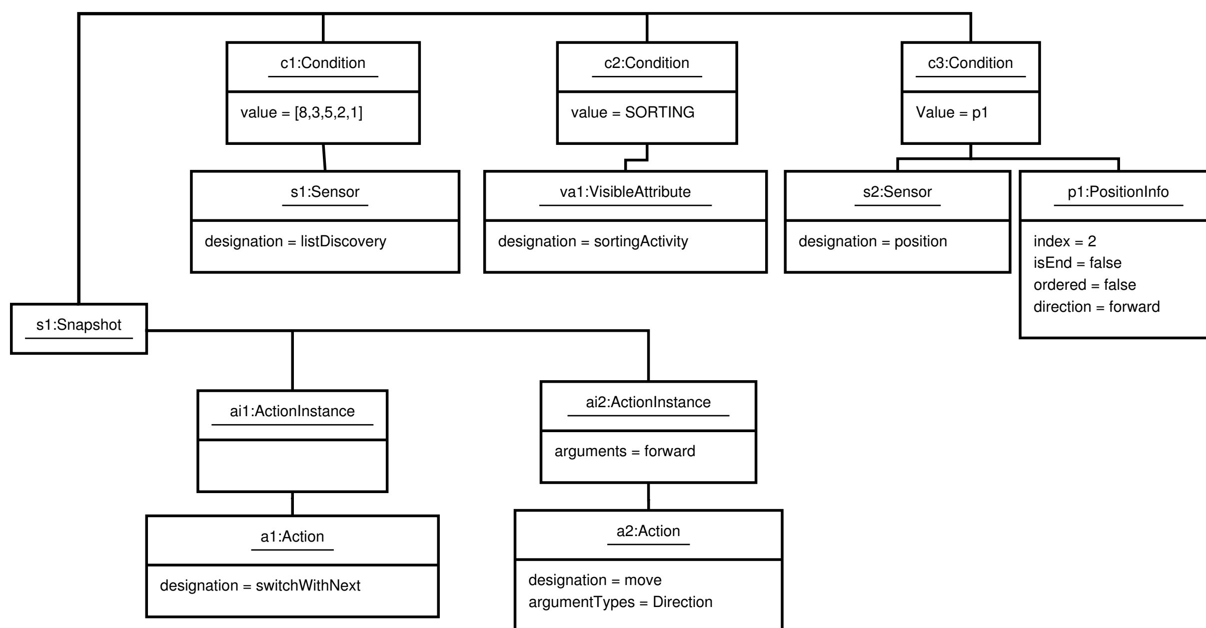


Figure 3.5: The object diagram of an example of a snapshot

The snapshot in figure 3.5 shows that when the agent is facing the conditions $C1 \wedge C2 \wedge C3$ it executes the actions *switchWithNext* and *move*. The *move* action receives the argument *forward* which designates the direction on which the agent has to move. The conditions in the snapshot show the information acquired by the agent sensors and the instances of the visible attributes before the actions were executed. The conditions show the list discovered by the *listDiscovery*

sensor [8,3,5,2,1] (condition C1). The agent's position in the list, as acquired by the *position* sensor, is given by the object P1 (condition C3), which determines that the agent is facing the second element which is not the end of the list. Condition C3 also shows that list is not ordered and the agent is moving forwards. The conditions also show the agent was displaying *SORTING* on its *sortingActivity* visible attribute at that time (condition C2).

Another important innovation on the dynamic image is the ability to hold a history record of earlier snapshots for a limited period of time. This allows observers to acquire information on past snapshots, regarding a period before they started to observe the agent. Acquiring information from the history record allows observers to gather knowledge much faster, when compared with observing only the snapshots regarding the current activity because it is not necessary to wait for the agent to perform those actions. Since this process consumes large amounts of memory, the history record contains only a limited number of snapshots, which can be configured according to the needs.

The software image is complemented with an ontology that provides a universal understanding of the designations of its different kinds of elements (sensors, visible attributes and actions), as explained in the following section.

3.3 The Software Image Meta-ontology

Using ontologies to organize and discriminate the different kinds of atomic elements and tasks provides a universal understanding of the software image because the ontology provides a common ground for the different designations of these elements. Different agents following the same ontology can use the same designations for the same kinds of sensors, actions, visible attributes and tasks, which facilitates the comparison of the software images of those agents (see section 3.4.3). The ontology also allows the atomic elements to be associated to specific tasks, which provides agents with knowledge on which elements are required to perform a task (see section 4.3.1).

Another important aspect of the ontology is the possibility of creating relationships between two different designations of same kind of element, which opens the possibility of translations between different ontologies. The translation allows different elements of two different software images to be interpreted as mutually corresponding elements. It also opens the possibility of combining different designations of the same element on different software images.

The software image meta-ontology was created to facilitate the integration and translation of different ontologies that represent the knowledge about the agent elements and tasks. The meta-ontology defines the basic elements for the ontologies and the possible relationships between those elements. Figure 3.6 shows a representation of the software image meta-ontology. A complete definition of this meta-ontology can be seen in appendix B.

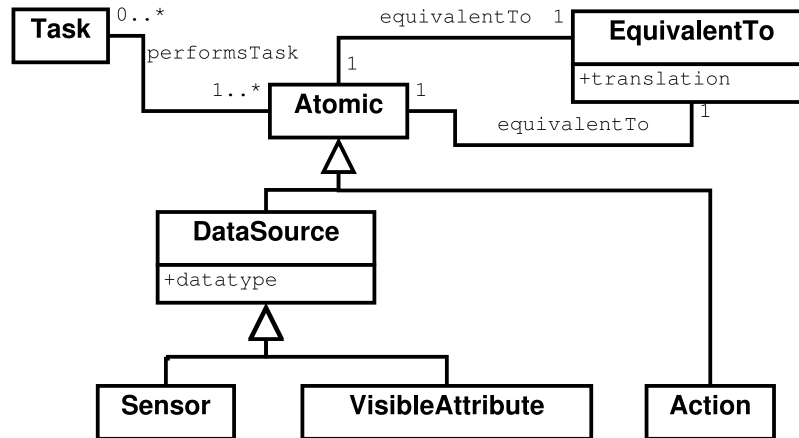


Figure 3.6: The software image meta-ontology

As figure 3.6 shows, all the atomic elements (sensors, visible attributes and actions) derive from the ontology *Atomic* class. Agent tasks are represented by the ontology *Task* class. A task can be associated to several atomic elements through the *performsTask* relationship. The relationship defines which atomic elements are necessary for accomplishing that task. The *performsTask* relationship also shows what tasks can be accomplished by a specific atomic element.

An atomic element can also have a relationship with another atomic element through the *equivalentTo* relationship, which allows two different atomic elements to be regarded as mutually corresponding. The *equivalentTo* relationship can be extended with a *EquivalentTo* association class, whose *translation* property points to a translation component.

The translation component is essential for relationships between two sensors or two visible attributes. It is capable of converting the information acquired by one of the sensors (or the information in one of the visible attributes) in the relationship to the information acquired by the other element. This conversion is important for understanding the information acquired by the equivalent sensor or in the equivalent visible attribute.

The sensors and visible attributes derive from the *DataSource* ontology class because they are the data-sources for the conditions (see section 3.2). The *DataSource* ontology class holds a *datatype* property that indicates the kind of data provided by the sensor or of the visible attribute.

It indicates the kind of information the sensor is capable of acquiring from the world or how an aspect of the internal state is represented in a visible attribute. Figure 3.7 shows an example of how the *datatype* property can be used.

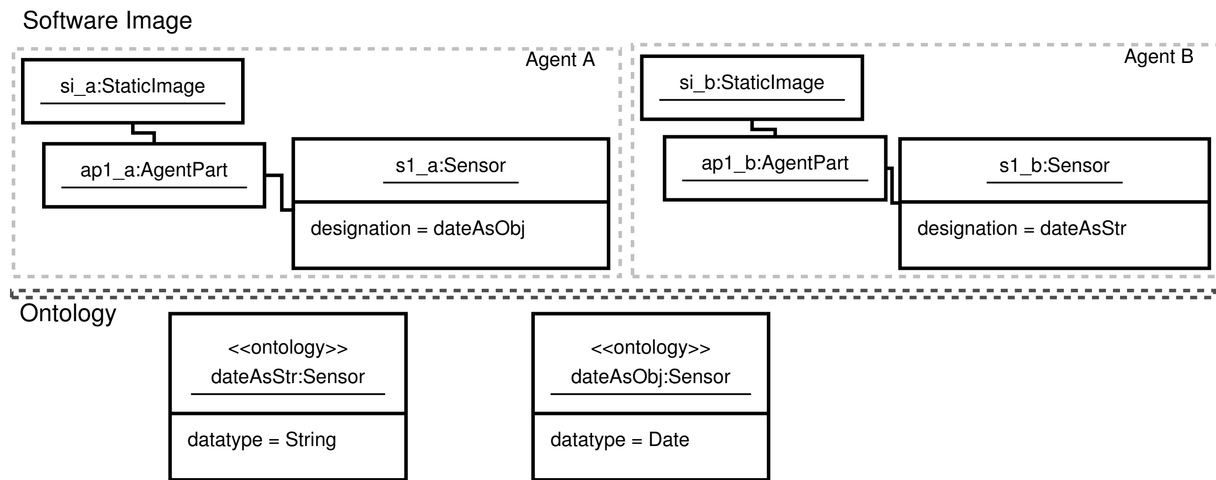


Figure 3.7: An example of use of the *datatype* property

Figure 3.7 shows part of the static images of two agents (agent A and agent B). Agent A has a sensor (*dateAsObj*) that acquires a date in a specific format from the environment and agent B has a sensor (*dateAsStr*) that acquires a date in another format from the environment. Even though both agents acquire the same kind of data from the environment, a date, their sensors have different designations because each agent has a different interpretation of the world. The *datatype* property of each designation shows how different are these interpretations. Agent A acquires the date as a *Date* object whereas agent B acquires the date as a *String* object.

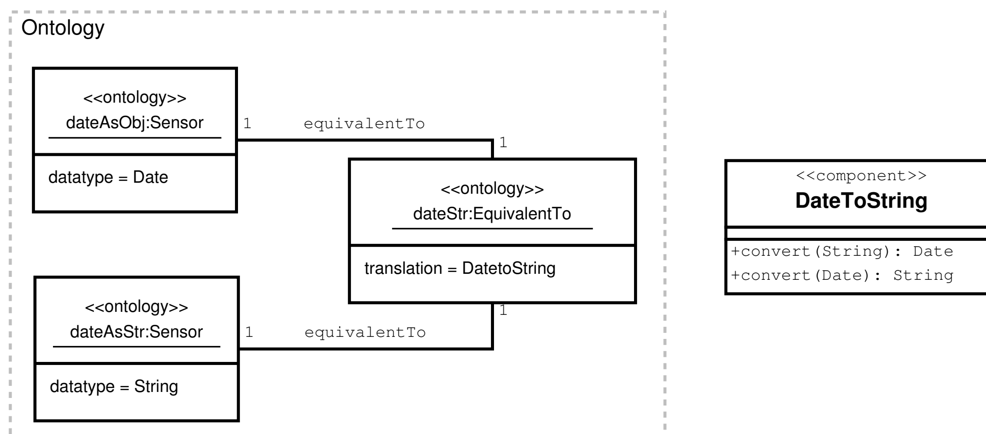


Figure 3.8: An example of use of the *equivalentTo* relationship

Under these conditions the two static images would be regarded as different when compared with each other because their sensors have different designations. However, there is a possibility

of making these two sensors mutually corresponding using an *equivalentTo* relationship. This relationship allows two different elements from two different software images to be regarded as mutually corresponding elements, which allows the two static images to be regarded as identical. Figure 3.8 shows how the *equivalentTo* relationship can be used in the ontology from the example in figure 3.7 to make the two sensors mutually corresponding.

As figure 3.8 shows, the *equivalentTo* relationship associates the *dateAsObj* sensor from agent A with the *dateAsStr* sensor from agent B. Thus, even though the sensors have different designations, they are regarded as corresponding when comparing the static images of the agents (see section 3.4.3). The figure also shows that the *equivalentTo* relationship is extended with an *EquivalentTo* association class (*dateStr*). The *DateToString* component shown in the *dateStr* association class is capable of converting a date from a *String* to a *Date* object and also in the opposite direction.

Thereby, when agent A acquires snapshots from agent B, the conditions concerning the information provided by the *dateAsStr* sensor are converted to *Date* objects, which is the kind of data provided by the *dateAsObj* sensor, using the *DateToString* component. The *DateToString* component allows agent A to observe and understand the conditions in agent B's perspective even when agent A has a different interpretation of the world. Section 5.2.5 shows the results of additional uses of the *equivalentTo* relationship.

Appendix B presents the ontology used for the example in figures 3.7 and 3.8.

The following section presents the computational approach for the software image

3.4 The Software for the Software Image

The computational approach for the software image provides the software which allows building, updating, comparing and discovering software images. The software for the software image also provides a mechanism for holding the conditions, that is, the information on the current state of the environment as acquired by the agent sensors and the instances of the visible attributes at a given moment (see section 3.2). An implementation of this software is accessible from a software versioning and revision control repository [1].

Figure 3.9 shows the way the functionalities and services provided by software developed for the software image are organized. The software for the software image was subjected to a

[1] https://github.com/coostax/lckosa-phd.git project SoftwareImageFramework

series of tests whose results are reported in section 5.1.

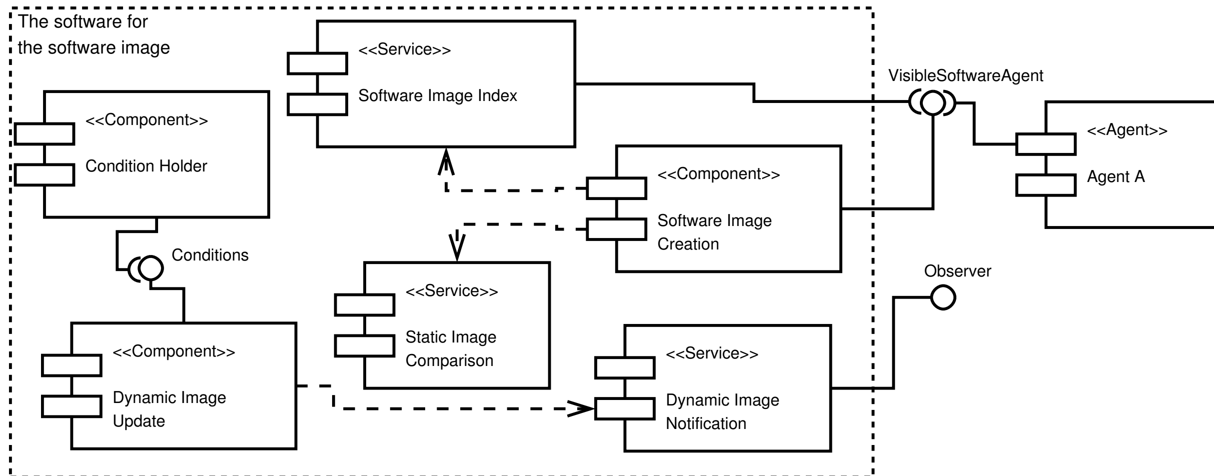


Figure 3.9: The functionalities provided by the software for the software image

As figure 3.9 shows, the software for the software image provides the instruments for building the software image (see section 3.4.1) in the *Software Image Creation*. The *Condition Holder* is responsible for acquiring and storing the conditions holding for the agent actions and the *Dynamic Image Update* holds the mechanisms that update the dynamic image with new snapshots (see section 3.4.2).

The *Static Image Comparison* holds the mechanisms for comparing agent software images (see section 3.4.3). The *Dynamic Image Notification* allows agents to register for notifications on new snapshots. The registered agents receive a notification whenever a new snapshot is displayed in the dynamic image (see section 3.4.3). Finally, the *Software Image Index* provides a shared repository where all registered software images can be accessed, thus facilitating the discovery of software images (see section 3.4.3).

Figure 3.9 also shows that the software created for the software image defines three different kinds of interfaces: the *VisibleSoftwareAgent* interface, the *Observer* interface and the *AgentConditions* interface. The *VisibleSoftwareAgent* interface provides access to the software image while the *Observer* interface allows the agents that are interested in observing the represented agent to subscribe for updates to the dynamic image (see section 3.4.3). The *AgentConditions* interface provides access to the conditions in the *Condition Holder*.

The agent software image and all the mentioned functionalities require only a minimal intervention in the agent original source code. Agent developers only need to annotate the elements that are going to be visible in the software image using code annotations. Figure 3.10 shows the seven different kinds of code annotations that were specifically defined for this purpose.

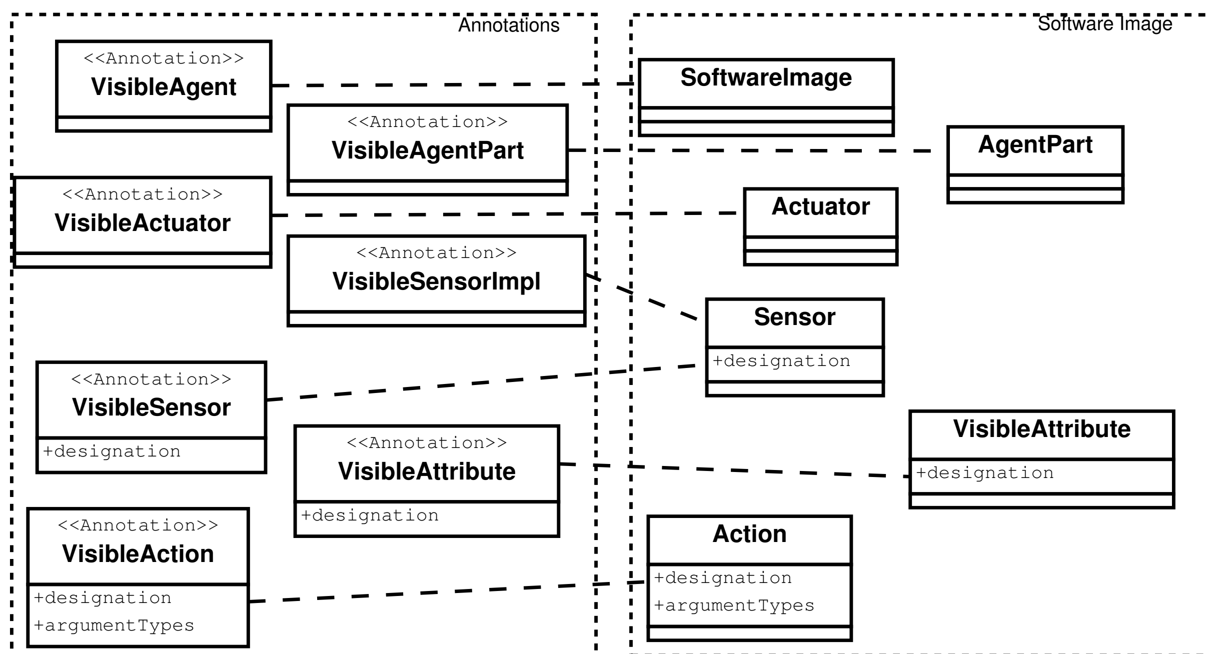


Figure 3.10: The annotations and their relation with the software image

The annotations presented in figure 3.10 enable the identification of the classes of the software agent and the methods and attributes of those classes that are essential for creating and updating the software image, as described in sections 3.4.1 and 3.4.2. As explained in section 3.2.1, the software image describes the software agent as a composition of agent parts, each of which can have sensors, visible attributes and actuators which in their turn provide the agent actions. The *VisibleAgent* and *VisibleAgentPart* annotations identify the implementation classes of the agent and of the agent parts. The class that implements the agent part can have sensor acquisition methods (annotated with *VisibleSensor*), action methods (annotated with *VisibleAction*) and attributes that represent important aspects of the agent's internal state, the visible attributes (annotated with *VisibleAttribute*).

When the class that implements the agent part (annotated with *VisibleAgentPart*) contains action methods, those actions are automatically associated to a default actuator. In addition, the class that implements the agent part can also be associated to other classes that implement a sensor mechanism (annotated with *VisibleSensorImpl*) or provide the agent with a set of actions, an actuator (annotated with *VisibleActuator*). Classes that implement a sensor mechanism must have a sensor acquisition method annotated with *VisibleSensor* for the sensor to be visible in the static image (see section 3.4.1).

Figure 3.10 also shows that the designation of an atomic element is assigned in the *designation* attribute of the annotation (*VisibleSensor*,*VisibleAction* and *VisibleAttribute*). For consis-

tency purposes, the designations may follow an ontology (see section 3.3). The *argumentTypes* attribute in the *VisibleAction* annotation assigns the type of arguments that can be used to invoke the action. It is also possible to assign a source that provides the arguments in the *argumentTypes* attribute, which allows invoking the action from generic invocation mechanisms. Figure 3.11 shows an example of how the *argumentTypes* attribute can be used for generic invocation.

```
<<@VisibleAgentPart>>
SomeAgentPart
-------------------------------------------------------------
 @VisibleSensor(designation=sensorOne)
+collectInfoOnOne(): TypeOne
 @VisibleSensor(designation=sensorTwo)
+collectInfoOnTwo(): TypeTwo
 @VisibleAction(designation=actionOne,argumentTypes=[sensorOne,
                sensorTwo])
+doSomething(d1:TypeOne,d2:TypeTwo)
```

Figure 3.11: An example of using the *argumentTypes* attribute for generic invocations

Figure 3.11 shows the implementation class of an agent part with two sensor methods and one action method. The methods annotated with *VisibleSensor* represent the sensors of the agent part and the method annotated with *VisibleAction* represents the action of the agent part. The example in figure 3.11 shows that the action *actionOne* is instantiated with two arguments. The *argumentTypes* attribute of the *VisibleAction* annotation indicates that the value for the first argument is provided by sensor *sensorOne* and the value for the second argument is provided by sensor *sensorTwo*. This means that, to invoke action *actionOne*, an invocation mechanism can use the information provided by sensors *sensorOne* and *sensorTwo* as the arguments for the action.

Besides annotations, the software developed for the software image uses other technologies such as code introspection and aspect oriented programming. The following sections describe the functionalities of the software developed for the software image in greater detail, showing the contributions provided by these technologies.

3.4.1 Software Image Creation

This section describes a generic process to automatically create the software image and the static image of an agent. The method initially chosen to build the static image (and therefore the software image) was based on the construction of abstract syntax trees (Neamtiu *et al.*, 2005) through code introspection. However, the representation provided by the abstract syntax

tree is too complex for the software image. One way of simplifying this information is to categorize it using annotations. Therefore, the static image of an agent is created by a domain independent process that uses code introspection to find annotated elements (classes, methods and attributes), in the agent's code, that represent the relevant features of the software agent to be included in the software image.

The annotations are placed in the agent's source code to identify the agent and agent part implementation classes, the sensor and action methods, and the visible attributes. The annotations are also placed in components that implement sensor mechanisms and that provide action methods. Through code introspection it is possible to access all the agent's components from the agent's main class. This way, the creation process is able to locate all the annotations in the agent's source code and build an appropriate static image for the software agent with the information provided by the annotations. A general description of the creation process, which starts whenever the agent's main class is initialized, is shown in algorithm 1.

Algorithm 1 Creating the software image of an agent whose main class is $agentClass$

function BUILDSOFTWAREIMAGE($agentClass$)
  init $staticImage$ ▷ the agent's static image
  ▷ regard $agentClass$ as an agent part
  $part \leftarrow$ BUILDPARTIMAGE($agentClass$)
  add $part$ to $staticImage$
  ▷ find other agent parts in $agentClass$ attributes
  $attList \leftarrow$ the list of attributes of $agentClass$
  for all $att \leftarrow$ attribute in $attList$ **do**
    $class \leftarrow$ the implementation class of att
    if $class$ has VisibleAgentPart annotation **then**
      $part \leftarrow$ BUILDPARTIMAGE($class$)
      add $part$ to $staticImage$
    end if
  end for
  init $softwareImage$ with $staticImage$
  return $softwareImage$
end function

The process described in algorithm 1 shows that the static image creation process is also responsible for initializing the agent's software image, since it is called when the agent is initialized. Therefore, this process guarantees that all agents have a software image from the moment they are initialized, as proven by the tests described in section 5.1.

Algorithm 2 Creating the static image for the agent part declared in $class$ (part1)

```
function BUILDPARTIMAGE(class)
    init part                                        ▷ the static image of the agent part
                        ▷ find elements in attributes and attribute implementation classes
    for all att ← attribute of class do
        subClass ← the implementation class of att
        if att has VisibleAttribute annotation then
                                                              ▷ att is a visible attribute
            vAtt ← data from VisibleAttribute annotation
            add vAtt to part
        else if subClass has VisibleSensorImpl annotation then
            init sensor                                  ▷ subClass implements a sensor
            init method              ▷ check if subClass has a sensor acquisition method
            for all m ← method of subClass do
                if m has VisibleSensor annotation then
                    method = m
                    break cycle
                end if
            end for
            if method is not null then
                sensor ← data from VisibleSensor annotation
                add sensor to part
            end if
        else if subClass has VisibleActuator annotation then
            init actuator                             ▷ subClass implements an actuator
            for all m ← method of subClass do
                if m has VisibleAction annotation then
                    action ← data from VisibleAction annotation
                    add action to actuator
                end if
            end for
            add actuator to part
        else if subClass has VisibleAgentPart annotation then
                                                    ▷ subClass implements an agent part
            subPart ← BUILDPARTIMAGE(subClass)
            add subPart to part
        end if
    end for
                                                            ▷ continues in algorithm 3
```

Algorithm 1 also shows that the process considers the agent's main class as an agent part, with sensors, visible attributes, actuators and actions. The algorithm shows how the static image is created from a collection of agent parts. The process of creating the agent parts is presented in algorithm 2 and 3.

The first part of the process (algorithm 2) describes how the static image of an agent part is created from the part's class attributes and their implementation classes. The algorithm shows the way introspection is used on the attributes, declared in the part implementation class, to find the part's visible attributes, sensors, actuators and associated parts.

Algorithm 3 Creating the static image for the agent part declared in $class$ (part2)

▷ BuildPartImage($class$) - find static image elements in $class$ methods

```
    init partActuator
    for all meth ← the methods of class do
        if meth has VisibleSensor annotation then                    ▷ sensors
            partSensor ← data from VisibleSensor annotation
            add partSensor to part
        end if
        if meth has VisibleAction annotation then                    ▷ actions
            action ← data from VisibleAction annotation
            add action to partActuator
        end if
    end for
    if partActuator has actions then
        add partActuator to part
    end if
     return part
end function
```

The second part of the process (algorithm 3) describes how the static image is created from the part implementation class methods. It shows the way introspection is used on the methods declared in the part implementation class to find the part's additional sensors and actions.

The process of creating the static image provides a unique identifier to the created software image. This identifier distinguishes a software image from other software images. After being created, the agent software image is automatically registered in the *software image indexing*, as explained in section 3.4.3.

For the process to create an appropriate software image, agent developers have to pay attention to a simple rule. They must ensure that there are explicit links between the visible elements and their parent objects, which can be achieved through class attributes. For example, an agent part can be instantiated in an attribute in the agent's main class and the sensor implementation

class can be instantiated in an attribute in the corresponding agent part. Using these explicit links facilitates the introspection of the necessary elements from the agent's main class.

When considering a worst-case scenario, the time required for the software image creation algorithm for a software image with n elements and depth a is $T_{max}(n) = 2a^n + 5n^3 + 7n^2 + 3n + 5$. For simplification purposes, the elementary unit of measurement regards only the operations on the software image or on any of its elements. The time expended by each operation is considered to be greater or equal than the time taken when performing any of the operations represented by the elementary unit of measurement.

Using the big-O notation it is possible to determine that $T_{max}(n)$ is $O(a^n)$ and therefore the software image creation process has an exponential time complexity a^n. If ability to search for agent parts inside agent parts (see section 3.2) is removed from the process, the total time becomes $T_{max}(n) = 5n^3 + 7n^2 + 3n + 5$, which is $O(n^3)$ and the algorithm's time complexity is lowered to cubic n^3. This shows that the ability to have several layers of agent parts inside other agent parts is the factor that increases the complexity of the software image creation process, which is proved by the experimental results in section 5.1.1.

The process described in algorithm 1 creates a static image that most closely resembles the agent structure. Software agents are not required to use this process to create their software image. Other processes can be used to create the software image as long as it represents the agent and is accessible to itself and to other agents interested in observing it.

The static image represents the static aspects of the agent, its components and their structural relations. Besides the static image, the software image is also composed of the dynamic image, which changes with time. The dynamic image needs to be updated with the agent's actions and the conditions holding when the agent decided to execute those actions. The next section describes the process responsible for updating this information.

3.4.2 Updating the Dynamic Image

This section describes two functionalities provided by the software created for the software image. The *Dynamic Image Update* (see figure 3.9) presents a process for creating snapshots, updating the dynamic image and managing the history record. The *Condition Holder* (see figure 3.9) maintains a copy of the conditions holding before the actions were executed by the agent, which consist of the state of the world as it is perceived by the agent sensors and the instances of the visible attributes.

The dynamic image is responsible for displaying snapshots of the agent activity (see section 3.2.2). It allows other agents to observe the actions executed by the represented agent while it is performing a task. Tangible entities take advantage of the laws and properties of the physical world to make their actions automatically observable to others. Following this concept, the *Dynamic Image Update* tries to do the same thing with the software image. This functionality takes the burden of making the snapshots of the agent activity visible to other agents.

As described in section 3.2.2, the snapshot holds information on the conditions (the state of the world as it is perceived by the agent sensors and the instances of its visible attributes) and the actions that were executed since the moment these conditions were determined (when the agent updated its perception of the world) until the moment in which the agent updates its perception of the world once again. A snapshot can be seen as a positive training example (used for training learning algorithms) consisting of the actions executed by the agent and the ideal conditions in which those actions may be executed.

A new snapshot is created each time the agent updates its perception of the world and executes actions. The agent sensors acquire information on the state of the world, thus creating the agent's perception of the world. Each time the agent updates its perception of the world, it uses this information, along with important aspects of its internal state (the visible attributes), to select actions and execute them. Figure 3.12 presents a graphical description of how snapshots are created.

Figure 3.12 shows that creating snapshots requires the functionalities of both the *Condition Holder* and the *Dynamic Image Update*. The figure also shows that the several activities of snapshot creation (distinguished by a grey background) are triggered by aspect oriented programming (AOP). As figure 3.12 shows the *Condition Holder* plays an important role in creating snapshots. It is responsible for maintaining a copy of the information acquired from the sensors and of the instances of the visible attributes, the conditions for the snapshot. This is the only way to ensure that this information is not modified by any internal process of the agent.

As figure 3.12 shows, the conditions are collected when the agent updates its perception of the world and acquires data from its sensors. Each time the agent acquires data from a sensor, the *Condition Holder* stores a copy of this information. When the agent finishes updating its perception of the world, the *Condition Holder* stores a copy of the instances of the visible attributes at that given moment. After the agent selects the actions to execute, the snapshot is initialized with the conditions stored in the *Condition Holder*.

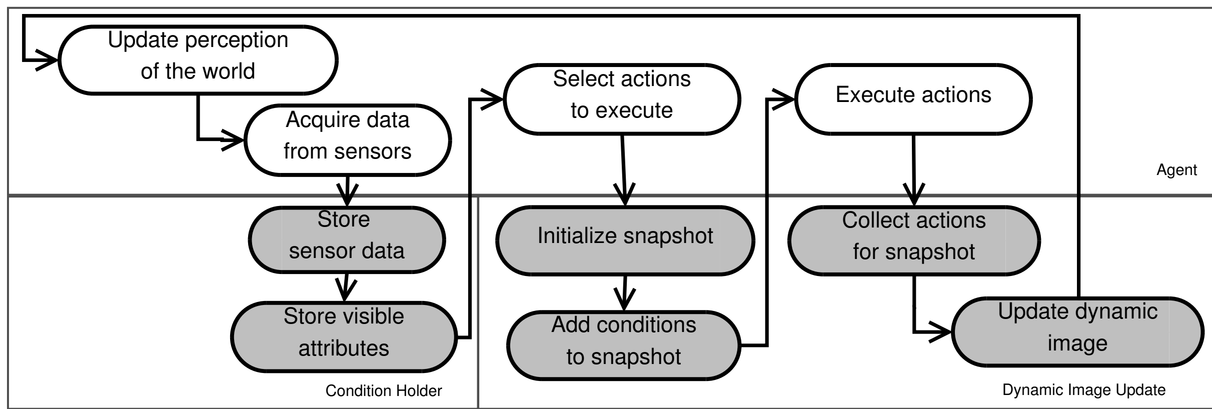


Figure 3.12: Creating snapshots and updating the dynamic image

When the agent starts executing the actions, that is, calling the annotated action methods (see section 3.4.1), aspect oriented programming records the actions, as they are instantiated by the agent, in the snapshot. After the agent has executed all actions and before it starts updating its perception of the world, the snapshot is finalized and the dynamic image is updated with the new snapshot.

Each time the dynamic image is updated with a new snapshot, the previous snapshot is moved to the history record. If the history record's size limit is reached, the oldest snapshot is deleted to make space for new snapshots. History data is organized in a first-in first-out (FIFO) approach, ensuring an uninterrupted sequence of snapshots from a given moment in the past up to the present.

An important aspect to take into account when defining the history record's size limit is the consumption of memory space. In a worst-case scenario, the amount of memory required to store n elements in the history record is $f_{max}(n) = 2n^2$, where n represents either a condition or an action instance. For simplification purposes the memory required to store n is considered to be greater or equal than the amount of memory required to store a condition or an action instance. Using the big-O notation it is possible to determine that $f_{max}(n)$ is $O(n^2)$ and therefore the history record has a quadratic memory complexity n^2.

Creating and updating the software image are two important features for a general purpose software image. However, the proposed software image is focused on learning by observation. The following section shows additional functionalities that are shaped for learning by observation: the discovery, comparison and observation of software images.

3.4.3 Discovering and Observing the Software Image

This section describes three services provided by the software for the software image (see figure 3.3) and a new protocol for observing the agent. The first service, the *Software Image Index*, consists of a shared medium where all software images can be registered. The second service, the *Static Image Comparison*, allows agents to compare their software image with the software images of other agents. The third service, the *Dynamic Image Notification*, notifies agents whenever new snapshots are available in the dynamic image.

To facilitate the discovery of agent software images, the proposed software image provides a discovery service, the *Software Image Index*. This service allows software agents to register their software images in a shared medium so that other agents are able to find them. The shared medium holds pointers for the registered software images that can be examined by any agent that wants to observe software images. The index service also allows software agents to unregister their software images, making them unavailable for discovery.

The index service is automatically initialized when using the *Software Image Creation* to create the software images (see section 3.4.1). The *Software Image Creation* automatically registers all the software images in the index service. If another process of creating software images is adopted, the index service provides the necessary instruments for initializing the service and registering the software images.

The software images discovered through the *Software Image Index* can be compared with the software image of the agent that is discovering them using the *Static Image Comparison*. This service determines the constituents and capabilities, described in the static image, that are identical in both agents. Knowing what capabilities and consituents are identical is important for determining the extent to which two agents are identical, which is an essential aspect for learning by observation as explained in section 4.3.

The process of comparing software images only uses the static image, since it contains all the necessary information on the agent's constituents and capabilities: what kind of parts the agent has, how those parts understand the world and what actions they can perform. The static image comparison is a recursive process, based on tree comparison. The agent's static image is regarded as a tree. Each agent part is a sub-tree of the static image. The part's sensors and visible attributes are leafs of the part sub-tree. Each part actuator is a sub-tree of the part whose leafs represent the actions provided by that actuator.

The atomic elements of the static image (sensors, actions and visible attributes) are com-

pared by their designation. The elements with the same designation are regarded by the comparison process as being identical. As section 3.3 shows, the designations of the atomic elements can be represented in an ontology. The ontology defines an additional relationship, the *equivalentTo* relationship, that allows two different elements of two software images to be interpreted as mutually corresponding elements. Section 5.1.2 shows the results of the tests performed to the *Static Image Comparison* service.

The *Dynamic Image Notification* provides a new protocol for observing the represented agent. It implements the observer design pattern (Freeman *et al.*, 2004) to notify other agents that a new snapshot has been created in the dynamic image of the represented agent. The agents that want to receive these notifications must subscribe them using the *O*bserver interface (see figure 3.9). The notifications allow the agents to know when they can acquire a new snapshot from the dynamic image.

Chapter 4

Learning by Observation

This chapter describes the proposed approach regarding learning by observation in software agents. An overview of the proposed approach is presented in (Costa & Botelho, 2012). Learning by observation is one of the most commonly used learning techniques in humans and superior mammals (see section 2.2.1). It consists of building control knowledge from the information acquired by observation. It is similar to the supervised learning techniques with the exception that the information acquired is about the executed actions and it only contains positive examples.

For learning by observation to be possible, software agents need to see themselves or other agents in the same way as tangible entities see each other (see section 2.1). To get around this problem, a universally accessible software image was developed in this research (see chapter 3). Equipped with this software image, a software agent is able to ”see” a representation of itself and of the other agents. The software image describes the static structures of the agent and the executed actions and the conditions holding when those actions were executed. When agents observe the expert they are effectively acquiring information on the actions performed by that expert while it executes a task and on the conditions holding when those actions were selected for execution.

The proposed approach presents a global view of the learning process which follows Bandura's definition of learning by observation (Bandura, 1977). The proposed learning process comprises seven activities that not always happen in strict sequence:

1. Discovery of an expert agent from which it is potentially possible to learn;

2. Observation of the selected expert agent and acquisition of information regarding the

expert actions and conditions holding for those actions;

3. Storage of the information acquired from the expert in the agent's memory
4. Learning new control knowledge to propose actions;
5. Determining the current holding conditions;
6. Execution of the actions proposed by the new control knowledge to perform the agent task; and
7. Evaluation of the agent progress

The agent may be in one of two states regarding the learning by observation process: the learning state and the execution state. When agents are in the learning state, their only objective is to observe experts and acquire new knowledge from them. When agents are in the execution state, their only objective is to apply the acquired knowledge. In each of these states, the agent will have access to only a subset of the activities of the learning process. Figure 4.1 shows the activities available to each state.

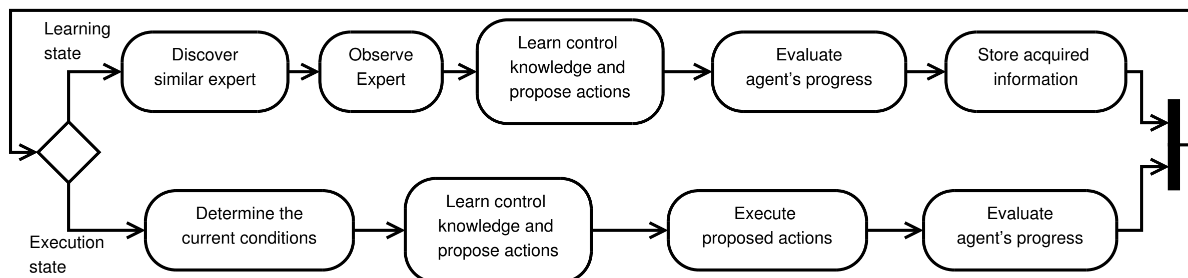


Figure 4.1: The activities of the learning process on each state

As figure 4.1 shows, the possible activities of the learning state concern discovering experts from which it is potentially possible to learn (see section 4.3.1) and observing the discovered expert, that is, acquire snapshots from its dynamic image (see section 4.3.2). Before storing the information from the acquired snapshot (see section 4.4), the agent tests its ability to perform the task it is observing. It does this by learning new control knowledge, from the information already existing in its memory, and proposing actions for the conditions in the snapshot (see section 4.5). The agent's evaluation compares the proposed actions with the actions in the observed snapshot to determine if the agent is able to propose the same actions as the observed expert when facing the same conditions (see section 4.6).

The possible activities of the execution state concern applying the agent's knowledge to execute a new task. The agent does this by acquiring the current state of the world from its sensors and combining it with the relevant aspects of its internal state (which are manifested in the visible attributes). Together this comprises the current conditions. The agent then learns new control knowledge from the information in its memory and proposes actions for the current conditions (see section 4.5). The proposed actions are executed by the agent actuators and evaluated (see section 4.6).

Two distinct methods of learning new control knowledge from the information acquired by observation, the recall and the classification methods, were specifically developed for the approach (see section 4.5). They were inspired on the two most used algorithms for learning by observation. The two methods are combined to present a single solution for apprentice agents, which increases the agent's ability to adapt to different situations. Agents are able to perform the observed task when facing the same conditions as the experts and also a similar task when facing different conditions.

Although the approach does not define any explicit mechanisms to motivate the agent to learn by observing others, this motivation arises when the agent discovers that it cannot yet perform a task. The agent's evaluation, described in section 4.6, is responsible for monitoring the agent's ability to perform a task by testing the agents capability of proposing the correct actions over time.

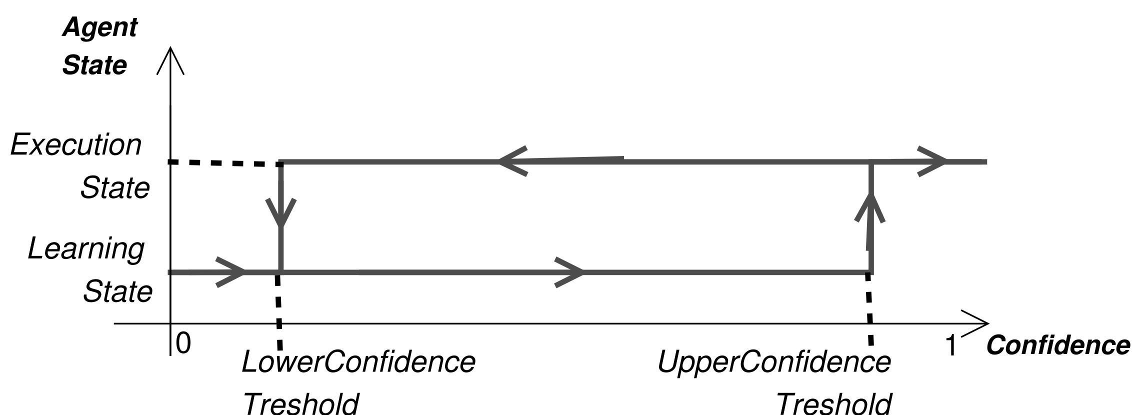


Figure 4.2: The influence of the confidence thresholds in state transition

The result of this monitoring is a measure of the agent confidence on the learnt knowledge, the confidence value. The agent will be on the learning or on the execution state depending on its confidence regarding the acquired knowledge. Two configurable thresholds, the UPPERCONFIDENCETHRESHOLD and the LOWERCONFIDENCETHRESHOLD determine the values at

which the agent changes to the learning or to the execution state. Figure 4.2 shows how the thresholds affect the transition between these two states.

As Figure 4.2 shows, the LOWERCONFIDENCETHRESHOLD defines the value at which the agent stops being confident on its knowledge and becomes motivated to learn more, thus changing to the learning state. The UPPERCONFIDENCETHRESHOLD defines the value at which the agent has enough confidence to become motivated to use the acquired knowledge, thus changing to the execution state. Section 5.2.3 shows how the values selected for these thresholds influence the agent's performance.

The agent's evaluation takes advantage of a special property of the proposed approach, the ability to mimic the behaviour of mirror neurons (see section 4.1), which allows the agent to propose actions both in the execution and in the learning state. The proposed approach also extends the agent capabilities beyond learning by observation by allowing external entities such as teachers and other evaluators to provide feedback about the executed actions (see section 4.6).

4.1 The Mirror Properties of the Learning Process

This section shows the way the proposed approach to learning by observation is able to mimic the behaviour of mirror neurons. It explains why mimicking this behaviour is important in learning by observation and how this ability is helpful for determining the agent's confidence on the learnt knowledge.

Section 2.2.4 shows that several approaches for learning by observation try to mimic the behaviour of the mirror neurons, especially because they allow an agent to feel almost like it is the one performing the actions it observes. The mirror neurons, which are active both when agents are observing and executing the actions, associate the concepts of actions with the actuators that execute them. This allows the agents to immediately recognize which of their actuators are able to execute the actions being observed. It also allows the agents to know which of their actuators execute the actions they decide to execute.

In the proposed approach, a similar effect is achieved by differentiating the description of an action from its execution. Both the observation of an action executed by another and the action executed by the agent are associated to the same description, which allows both the observation and the execution of an action to evoke the same representation. Using descriptions of the

actions is essential for making the actions of software agents visible to others. When a software agent observes another agent performing an action it is actually acquiring a description of the action being performed, which is generated by the software image (see chapter 3).

The agent uses the descriptions to reason about the actions and only executes them when necessary, which allows the agent to test its skills while learning. The agent realizes if it is capable (or not) of making the same decisions as the expert by comparing the actions it proposes, with the observed actions. The evaluation uses the result of this comparison to decide how the agent's confidence changes through time (see section 4.6).

4.2 Architectural Components of the Learning Process

The learning process is accomplished by an agent architecture whose implementation is accessible from a software versioning and revision control repository [1]. When combined with the software image (see chapter 3), the agent architecture provides all the necessary functionalities for learning control knowledge by observing the actions executed by experts while they perform a task.

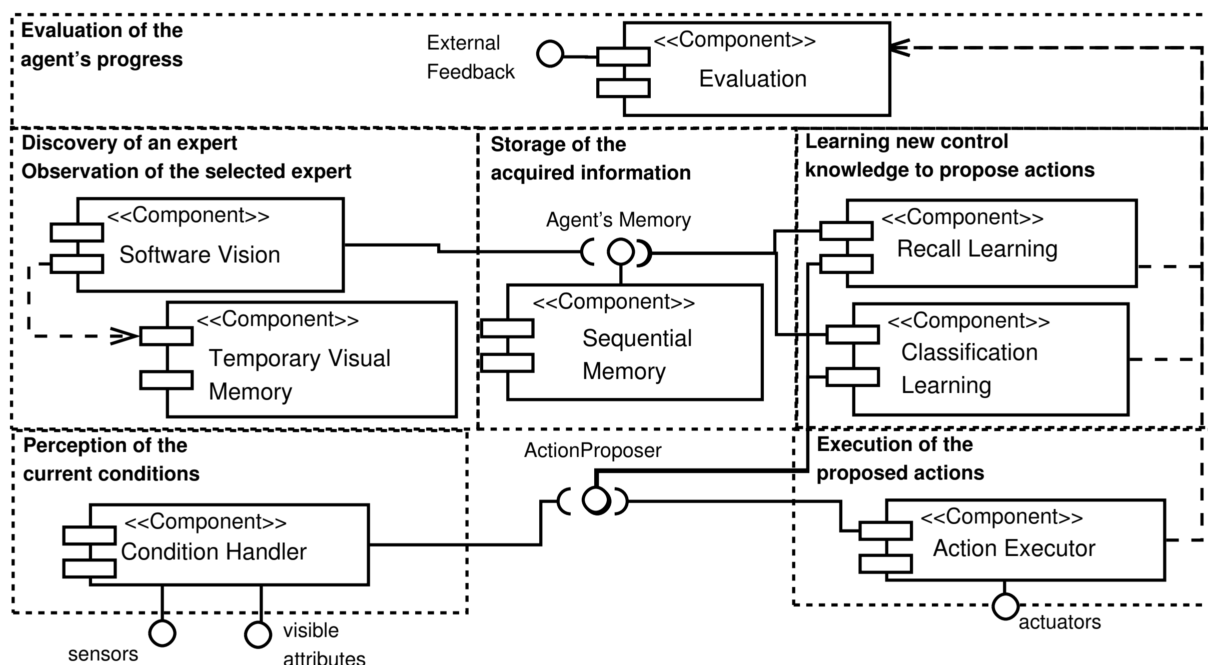


Figure 4.3: The components of the agent architecture

[1] https://github.com/coostax/lckosa-phd.git project LbOFramework

Figure 4.3 describes the agent architecture which follows a modular design where each module addresses specific activities of the learning process. The *Software Vision* component addresses the discovery of experts from which it is potentially possible to learn (see section 4.3.1) and the observation of the selected expert (see section 4.3.2). It uses the *Temporary Visual Memory* as a buffer for observation that allows the agent to handle the observed information at a different rate from which it is observed.

The *Sequential Memory*, the *Recall Learning* and the *Classification Learning* components define the learning algorithm of the proposed approach. The *Sequential Memory* component holds the mechanisms for storing the information acquired from observing the expert (see section 4.4). It is capable of holding the sequence on which the snapshots were observed.

The *Recall Learning* and *Classification Learning* components hold the mechanisms for learning new control knowledge from the information in the agent's memory and propose actions for the current conditions (see section 4.5). They provide the recall and the classification methods of learning.

The *Condition Handler* is responsible for determining the currently holding conditions. It holds the mechanisms that acquire the information that is necessary for the conditions and provides the interfaces for the agent sensors and visible attributes. When the agent is in the execution state, it uses the *Recall Learning* and *Classification Learning* components to propose actions for the current conditions.

The proposed actions are executed by the *Action Execution* component. This component links to the agent actuators that turn the descriptions of the actions in their effective execution.

The *Evaluation* component addresses the evaluation of the agent progress (see section 4.6). It holds the mechanisms for testing the agent's knowledge both when learning and when applying its knowledge. It also provides the interfaces for receiving external feedback from specialized experts and other kinds of evaluators.

The following sections describe the activities of the learning process in detail.

4.3 Discovering and Observing Software Agents

This section describes the capabilities of the software vision, which enables agents to discover experts and observe the actions they execute while performing a task. The section also shows the way the historic information contributes to a rapid increase of the agent's knowledge.

Agents are able to discover and observe experts only when in the learning state. Although it would be possible for an agent to observe experts and at the same time to perform a task, it would be harder to determine the influence of the newly acquired knowledge in the agent performance. Therefore, in the proposed approach, when agents are learning they are only focused on observing experts.

The process of discovering an expert and observing it is referred to as the observation period. This process is cyclical meaning that apprentices may go through several observation periods when in the learning state. In each observation period, the apprentice focuses on a single expert, meaning that it only collects snapshots from that expert's dynamic image. Figure 4.4 describes the activities that take place in the observation period.

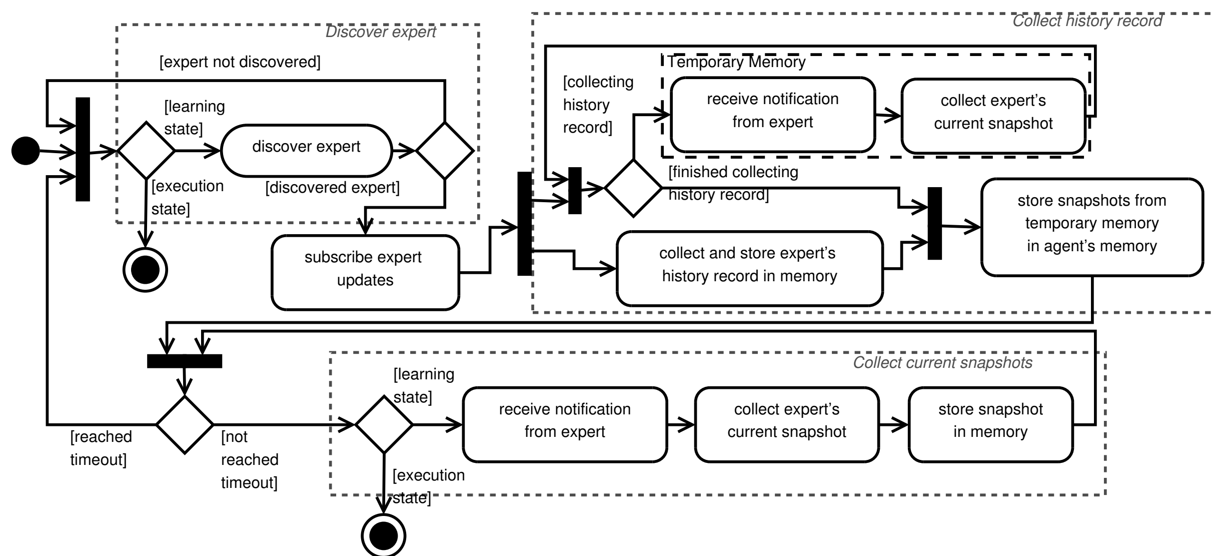


Figure 4.4: The process of discovering and observing experts

As figure 4.4 shows, the observation period comprises three main activities that take place in sequence: Discover an expert, collect the history record and collect the snapshots concerning the current activity. The length of an observation period is controlled by a timeout, after which a new period is started if the agent is still in the learning state. The observation period can also be terminated when the agent switches to the execution state, ceasing all learning activities.

The timeout for the observation period is defined by a configurable setting whose configuration depends on the application requirements. The shorter the observation period the greater the variety of experts observed, because the agent is encouraged to discover a different expert each time a new observation period starts. Observing different experts enhances the apprentice's knowledge, since these experts might provide a different perspective on the task to learn.

The different perspectives increase the agent's knowledge with conditions that were never

faced by any of the previously observed agents or with a different approach to the task to learn. The agent's memory is able to store different approaches for the task without conflicting with the existing knowledge because of the way this information is stored (see section 4.4.2).

Another important quality of the diversification of the observed experts arises when experts are not providing useful information, like for example, when the expert is doing nothing for a long period of time. When the agent faces this kind of situation, the current observation period is terminated, forcing the agent to discover another expert to observe. Thus, the agent has the chance of finding another expert that might provide useful information.

The following sections describe the process of discovering an expert, subscribing for expert updates and collecting snapshots from the expert.

4.3.1 Discovering and Selecting Experts to Observe

This section shows how experts are discovered, what kinds of experts are appropriate for an agent to learn by observation and how this expert is distinguished from the others. To discover other agents with software image, the proposed approach provides a discovery service, the *software image index*, which is described in section 3.4.3. This service provides a shared medium where software agents can find the software images of other agents.

Before starting to observe an expert, the agent must know if it is potentially possible to learn by observing that particular expert. When it is not possible to determine what agent structures are necessary for the task to learn (see section 3.3), the agents follow Bandura's social learning theory (Bandura, 1977) and learn by observing a similar expert, that is, an expert whose static image has the same structure and the same atomic elements as the agent's static image.

The agent uses the *Static Image Comparison* service of the software developed for the software image (see section 3.4.3) to compare its static image SI_{agent} with the static image of the expert SI_{expert}. If both images match, $SI_{agent} = SI_{expert}$, the expert can be observed because the agent is able to immediately recognize the observed actions and conditions (see section 3.2.2), solving most of the correspondence problems (Alissandrakis *et al.*, 2002; Argall *et al.*, 2009).

When using an ontology to describe the knowledge on the software image (see section 3.3) it is possible to determine which structures and abilities are necessary to perform a task T. As explained in section 3.3, the *performsTask* relationship determines which elements are necessary for carrying out the task. A partial static image, si_T, is created from the elements that

are necessary to perform the task. The concept of an expert from which it is potentially possible to learn is extended, allowing agents to observe an expert as long as the intersection of their software images contains those structures and abilities, $(SI_{agent} \cap SI_{expert}) \subseteq si_T$, as displayed in figure 4.5.

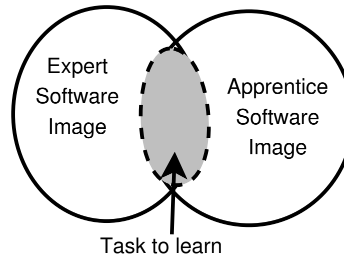


Figure 4.5: The concept of an observable expert

An important aspect to consider when discovering experts is that the observed expert might not be executing the actions for the task to be learnt. When the apprentice knows which elements are necessary to carry out the task, it is able to notice when the expert is not performing the actions for that task. Whenever this happens, the observation period is restarted for the agent to discover another expert. When it is not possible to know which elements are necessary to carry out the task, the time limitation in the observation period is a way of increasing the chances of finding an expert that is performing the actions for the task to learn while the agent is in the learning state.

Either way, the experts participating in the application scenarios where this approach was tested (see chapter 5) are programmed to only perform the actions for the task to learn. Another way of looking at this problem is assuming that agents have no specific goal or task to learn. This means that like in the initial stages of human development (Meltzoff & Moore, 1977; Lopes & Santos-Victor, 2007), the apprentices observe any expert, as long as it has the same visible structures, regardless of what the expert is doing. Thus, the agents are open to all the knowledge they can obtain from experts.

After discovering an expert from which the apprentice can potentially learn, the expert's image unique identifier (see section 3.4.1) is compared with the identifiers of previously observed experts. This allows the apprentice to determine if that expert was observed before. The proposed approach favours the observation of different experts, thus if the expert was already observed, the discovery process proceeds, in hopes of finding another expert that has not yet been observed.

If the discovered expert was not previously observed or if any other experts were found, the

agent proceeds with the observation of the expert, which is described in the following section.

4.3.2 Observing the Selected Expert

After discovering an expert from which it is potentially possible to learn, the agent starts observing it. As section 3.4.3 describes, observation consists of acquiring snapshots of the expert's activity from its dynamic image. The acquired snapshots describe a relation between an optimal set of conditions ($C_1 \wedge \cdots \wedge Cn$) and a sequence of actions ($a_1 \wedge \cdots \wedge am$), $C_1 \wedge C_2 \wedge \cdots \wedge C_n \rightarrow a_1 \wedge \cdots \wedge am$.

Before starting to acquire snapshots from the expert's dynamic image, the agent subscribes the *dynamic image notification*, a service provided by the software developed for the software image (see section 3.4.3) that facilitates the process of observing the expert. The notifications determine the ideal moment for the agent to observe, which is when a new snapshot is created in the expert's dynamic image

The first snapshots the agent acquires from an expert are the ones stored in the history record. The agent acquires all the snapshots in the history record, which allows the agent to obtain a large set of experiences in a small amount of time. This is more efficient than other solutions such as searching the history record on demand to find relevant information for a particular problem. Acquiring all the history record not only ensures that the solution for a particular problem is found (if it really exists in the history) but also provides the apprentice with an increased amount of information that might be important to solve other problems in the future.

Acquiring the history record at the beginning of observation provides the agent with an uninterrupted sequence of snapshots from a given moment in the past until the current moment. To prevent this sequence from being broken with the current activity from the expert, the history snapshots are handled before the snapshots regarding the current activity. While the agent is handling the snapshots from the history record, the snapshots regarding the current activity are acquired and stored in a temporary memory, thereby preventing their loss.

The temporary memory functions as a buffer for observation because it allows the agent to handle snapshots at a different rate from which they are acquired. It also allows the agent to keep record of the expert's actions that might take place while it is reading the history. Therefore, from the agent's point of view, it is like starting to observe the expert from a given moment in the past.

The following section explains how the knowledge provided by the acquired snapshots (both current and historic) is stored in the agent's memory.

4.4 Storing the Information Acquired in Observation

This section shows the way the agent holds the information contained in the observed snapshots in its memory. It describes the agent's internal knowledge representation and the data-structure where this information is stored. In the proposed approach, the information acquired from observation is stored in the agent's memory which holds the sequence in which this information was observed and therefore the sequence of actions executed by the agent while it was performing that task.

The agent learns from the information stored in its memory using the learning methods described in section 4.5, which provide the agent with new control knowledge. This control knowledge is used to propose actions for the current conditions.

4.4.1 The Internal Knowledge Representation

The snapshots acquired in observation resemble positive training examples (condition-action pairs), each describing the actions to execute when facing a set of conditions (see section 3.2.2). When a snapshot is observed, the relation between the conditions and the sequence of actions is stored in the agent's memory as an *Experience*.

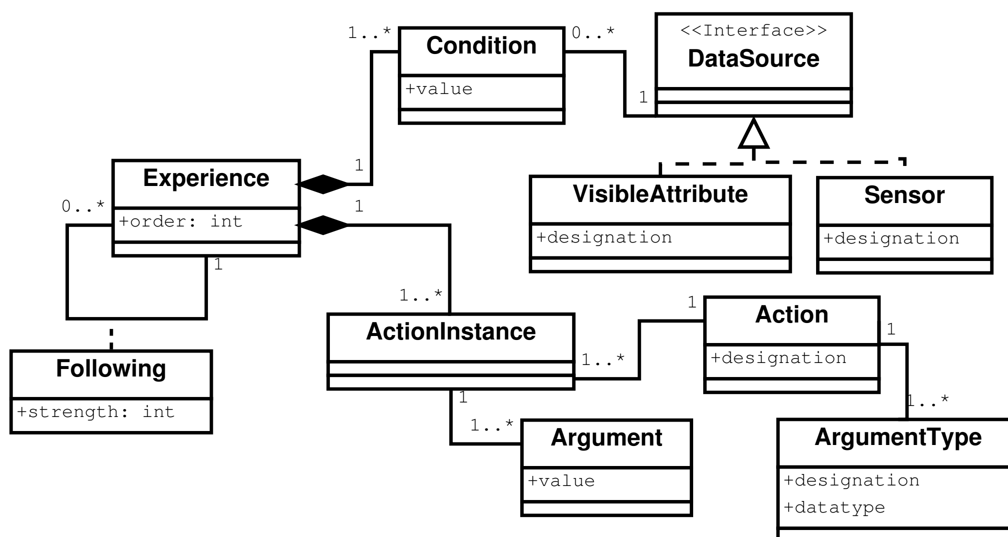


Figure 4.6: The EXPERIENCE and its relation with the snapshot

The *Experience* holds the knowledge on the actions the agent should execute when facing the observed conditions. It is created from the information contained in the observed snapshot. Figure 4.6 shows a representation of the *Experience*. The figure shows that, like the snapshot, the *Experience* holds a set of conditions and a set of action instances.

In the snapshot, the actions were performed by the expert and the conditions represent the information acquired by the expert's sensors and the instances of the expert's visible attributes. When the *Experience* is created, the actions from the observed snapshot are converted to the agent's own actions and the conditions are converted to the information acquired by the agent sensors and the instances of its visible attributes. The conversion uses the functionalities of the software image ontology described in section 3.3.

Given the properties of the proposed approach, the agent can determine which of its actuators provides the actions represented in the *Experience* because both the agent actions and the actions in the *Experience* use the same descriptions (see section 4.1). The agent is also capable of understanding the conditions in the *Experience* and determine the sensors and visible attributes that provide that information because the conditions are of the same kind as the information acquired by its sensors and in its visible attributes (see section 3.1).

Figure 4.6 also shows that the agent's memory is a tree of *Experiences* because each *Experience* can be linked to one or more *Experiences*, each representing the snapshot that was observed immediately after. Figure 4.7 shows a representation of the agent's memory.

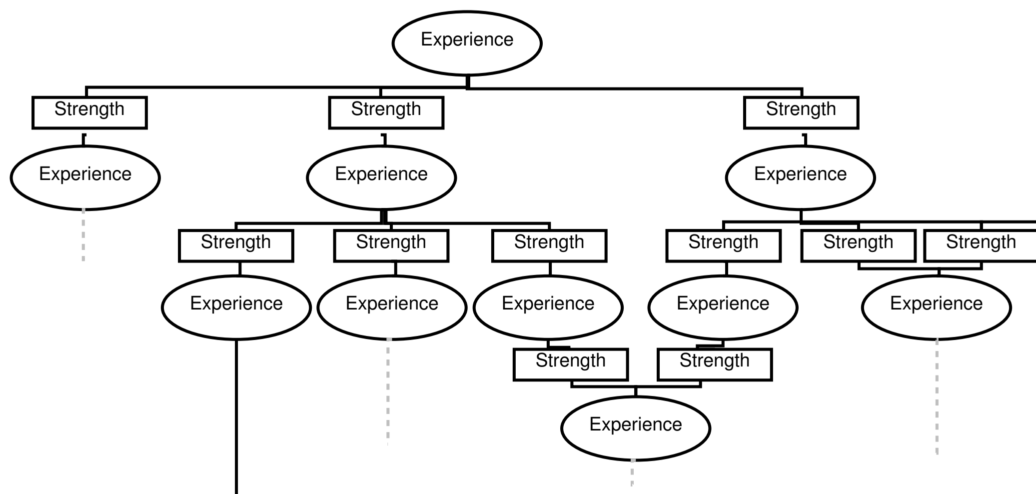


Figure 4.7: A representation of the structure holding the agent's memory

Using a tree structure to hold the agent's memory enables the sequence in which the snapshots were observed to be intrinsically preserved. Unlike the linear structures, which also intrinsically preserve the sequence in which the snapshots were observed, the tree structures fa-

cilitate the consolidation of the knowledge obtained when observing different experts because they store different approaches for the same task as alternative paths.

Figure 4.7 shows that the tree structure holding the agent's memory allows the agent's knowledge to be expressed as a decision tree, where each node of the tree is an *Experience*. Depending on the number of branches starting from the node, each *Experience* can be followed by one or more *Experience*, which adds the possibility of choosing which sequence to follow and therefore provide different alternatives for executing a task.

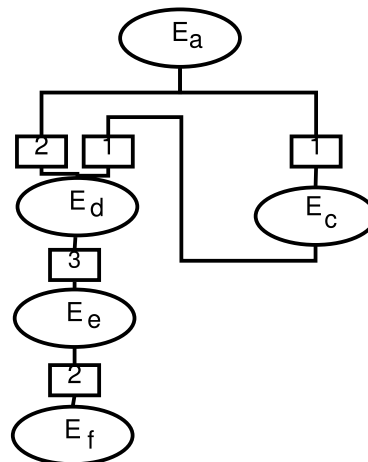


Figure 4.8: An example of the agent's memory

The relation between an *Experience* and the ones that follow it is characterized by a strength. The strength defines the absolute number of times the agent observed one *Experience* being followed by the other one. Figure 4.8 shows the agent's memory after observing three experts, each performing one of the sequences of snapshots $[S_a, S_c, S_d, S_e]$, $[S_a, S_d, S_e, S_f]$ and $[S_a, S_d, S_e, S_f]$. After observing the three experts, in the agent's memory E_a (the result of observing S_a) is followed by E_c (the result of observing S_c) with a strength of 1 and by E_d (the result of observing S_d) with a strength of 2.

The strength of the relation between an experience and its following plays an essential role on choosing the approach to follow when observing experts with different approaches for the same task. An expert has a different approach when it executes a different sequence of actions, which in turn leads to a different sequence of snapshots, to perform a task. The relations between an experience and the experiences that follow it are strengthened each time the agent observes an expert with the same approach to a task, that is, when it observes the same sequence of snapshots. Therefore, for the agent to choose an approach over another it has to observe several experts following that approach.

4.4.2 Storing Experiences in the Agent's Memory

This section shows the way the *Experiences* created from the snapshots observed on different experts are stored in the agent's memory and how the size of the conditions in the *Experiences* in memory is controlled. To simplify the methods of learning from the information in the agent's memory (see section 4.5), the size of the conditions in the *Experience* is the same as the size of the conditions acquired by the agent (the information on the state of the world as it is acquired by the agent sensors and the instances of the visible attributes). This allows the learning methods to directly compare the current conditions with the conditions in the *Experience*.

The size of the conditions in the *Experience* depends on the size of the conditions in the observed snapshot. If the agent was limited to observing experts with the same sensors and visible attributes there would be no concern regarding the conditions in the observed snapshots because all of them would have the same size as the conditions faced by the agent. However, as section 4.3.1 describes, the proposed approach extends the notion of an expert from which it is potentially possible to learn to allow agents to observe experts that might have a different number and different kinds of sensors and visible attributes.

The snapshots observed on those experts may have conditions with different sizes that provide information from sensors and visible attributes that the agent might not understand. The conditions may also miss information on certain aspects of the state of the world or on certain visible attributes that are not necessary for the task to learn. Therefore, the conditions in the *Experiences* obtained from those snapshots would have different size from the conditions in the *Experiences* in memory

To keep the size of the conditions constant for all the *Experiences* in memory, whenever the conditions in an observed snapshot have a different size or are of a different kind, the unknown or unnecessary information is discarded. This information is provided by sensors or visible attributes that do not exist in the agent and therefore are useless for learning. Information from the sensors and visible attributes that do not exist in the observed expert is also added to the conditions as EMPTY data. The EMPTY a special kind of data that represents information that is not important for deciding the actions to execute. It is identical to any condition of the same kind.

Given that agents can observe different experts, the sequence on which the snapshots are observed is broken, in the agent's memory, each time the agent starts observing another expert. When this happens, the snapshot observed before represents the activity of another expert and

therefore cannot be followed by the new snapshots. To prevent the fragmentation and duplication (multiple instances of identical *Experiences*) of the agent's memory, the process of storing a new *Experience* in the agent's memory compares the new *experience* with the ones existing in memory before storing it. Algorithm 4 shows how the process takes place.

Algorithm 4 Storing Experience $nexp$ in the agent's memory

Require: $nexp$ not from the first snapshot of expert ▷ It cannot be the first snapshot observed on that expert
Require: $prevexp$ to be the experience from the snapshot observed before
  function ADDEXPERIENCE($nexp$)
    $found \leftarrow false$
    for all $exp \leftarrow$ experiences in memory **do**
      if exp has the same conditions and the same action instances as $nexp$ **then** ▷ $nexp$ already in memory
        if exp is already following $prevexp$ **then**
          increase strength by one
        else
          add exp to the list of following experiences of $prevexp$ ▷ exp follows $prevexp$
        end if
        $found \leftarrow true$
        $prevexp \leftarrow exp$
        break cycle
      end if
    end for
    if $found$ is false **then** ▷ $nexp$ not in memory
      add $nexp$ to agent memory ▷ store $nexp$ in memory
      add $nexp$ to the list of following experiences of $prevexp$ ▷ $nexp$ follows $prevexp$
      $prevexp \leftarrow nexp$
    end if
  end function

The process described in algorithm 4 shows that an *Experience* is stored in the agent's memory only when no identical *Experiences* are found. If an identical *Experience* is found, the process only creates a connection between the *Experience* regarding the snapshot observed before and the identical *Experience* in memory. If the connection already exists the strength of that connection increases because the agent has observed the same sequence before.

The process in algorithm 4 shows how an *Experience* is stored when it is not obtained from the first snapshot observed on the expert. The only difference in storing an *Experience* from the first snapshot is the lack of a previous experience. Therefore, if an identical *Experience* is not found in the memory, there will not be a link from another experience to the newly stored one. The process is the same whether the *Experience* comes from a snapshot on the history record or

from a snapshot from the expert's current activity.

Considering a worst-case scenario, the time required to store a set of n elements in the agent's memory is $T_{max}(n) = n^2 + n + 3$, where n represents a condition or an action instance from an *Experience*. For simplification purposes, the elementary unit of measurement regards only the operations on *Experiences*, its conditions and action instances. The time it takes to perform an operation is considered to be greater or equal than the time it takes to perform any of the operations represented by the elementary unit.

Under the same worst-case scenario, the amount of memory required to store n elements in the agent's memory is $f_{max}(n) = 2n^2$, where n represents a condition or an action instance from an *Experience*. Using the big-O notation it is possible to determine that $T_{max}(n)$ is $O(n^2)$ and that $f_{max}(n)$ is $O(n^2)$. Thus, the process of storing an *Experience* in the agent's memory has a quadratic time complexity n^2 and a quadratic memory complexity n^2.

The knowledge stored in the agent's memory is used by the learning methods to propose actions for the current conditions. The following section explains these methods in detail.

4.5 Learning from the Information in the Agent's Memory

This section describes the learning algorithm of the proposed approach. It shows the way agents learn new control knowledge from the information acquired in observation and how this control knowledge is used to propose actions both in the learning and execution states. It presents the two methods for learning, the recall and classification methods, and explains how they are combined to present a single solution for proposing actions.

Several possibilities for the learning algorithm were experimented throughout the research. Many of these possibilities included implementations of known supervised learning algorithms. The results of the experiments in appendix C determined which of the supervised learning algorithms was best suited for the proposed approach and gave rise to the classification method of learning, which is an adaptation of the KStar algorithm to the kind of data used by the proposed approach, the snapshot (see section 4.5.2).

A second method of learning, the recall method (see section 4.5.1), was developed from the necessity of having a sequencing approach to learning because sequencing is one of the most used methods in the surveyed approaches to learning by observation (see section 2.2.2). The recall method was completely developed for this approach and is inspired on sequencing and

on sequence learning because it focuses on the same problems as sequencing.

The proposed learning algorithm is a combination of these two methods, which allows the agent to adapt to different situations. The recall and the classification methods use the information contained in the agent's memory (the tree of *Experiences* as explained in section 4.4.1) to propose actions for the current conditions. The two methods are used both when the agent is learning and when it is preparing to apply its knowledge and execute actions (see section 4.1).

The way the current conditions are acquired depends on whether the agent is in the learning or in the execution state. When the agent is in the learning state, the current conditions are the conditions in the observed snapshot. When the agent is in the execution state, the current conditions represent the current state of the environment, as it is acquired by the agent sensors, and the current instances of the agent visible attributes.

The recall and classification methods can be distinguished by the way they use the information in the agent's memory. The recall method proposes actions by analyzing the way the *Experiences* are connected with each other in the tree. The current conditions help to select the correct path in the tree (see section 4.5.1). The classification method categorizes the information in the observed data and determines which actions to perform according to the categories of the new problems (see section 4.5.2).

The recall method is best suited for learning tasks that require repetitive sequences of actions (for example, ordering a list). It is also suited for when the agent faces the same problems as the expert, like for example, in situations where the agent is following the expert. When facing the same problem, the current conditions are always found on an *Experience* in the agent's memory because the agent has observed an expert facing the same conditions. This is of extreme importance for determining where to start following the sequence of *Experiences*, as explained in section 4.5.1.

The classification method is best suited for dynamic environments where tasks do not depend on a specific sequence of actions (for example, a personal assistant learning how to interact with users). It is also important for when the agent faces a problem that is different from the problem faced by the expert. When the agent faces a different problem, the current conditions may not always be found on an *Experience* in the agent's memory because the agent has not observed an expert facing them. The classification method is resilient to this kind of situations and capable of proposing actions based on similar situations observed by the agent.

Both the recall and the classification methods provide several possibilities when proposing

actions, that is, they provide more than one solution for the actions to execute. To determine which of these solutions is the best suited, a RELIABILITY value is associated to each of them. The RELIABILITY of a solution is calculated by each method and ranges between zero and one, inclusive. It determines how reliable a solution is (zero - not reliable; one - fully reliable). Section 4.5.1 shows the way the RELIABILITY is calculated by the recall method and section 4.5.2 shows the way the the RELIABILITY is calculated by the classification method.

The recall and the classification methods of learning are combined in a single solution. The agent uses both methods to propose actions and to choose the best solution for the current situation. To help deciding which solution is best for the current situation, both the recall and the classification methods are associated to a WEIGHTFACTOR, whose initial and minimum value is zero. The WEIGHTFACTOR determines which of the methods is capable of proposing the most suitable action for the current situation.

The RELIABILITY of the proposed action is combined with the relative WEIGHTFACTOR of the method that proposed it, which results in the FINALRELIABILITY ($finalReliability = reliability \times relativeWeightFactor$). The way the FINALRELIABILITY is calculated gives the same proportion to the RELIABILITY and to the WEIGHTFACTOR. The action with the highest FINALRELIABILITY is the best one for the current situation.

The WEIGHTFACTORS change each time the agent's capacity of proposing actions is evaluated (see section 4.6). The WEIGHTFACTOR of a method increases if the action with the highest RELIABILITY proposed by that method is proven to be an appropriate choice by the internal evaluation. If the evaluation determines that the action was not appropriate, the WEIGHTFACTOR of the method decreases. As explained in section 4.6, the amount on which the WEIGHTFACTOR increases or decreases depends on the value of the RELIABILITY of the proposed action.

The time complexity of the proposed learning algorithm is given by the combination of the big-O notations for the recall method $T^{recall}_{max}(n)$ is $O(n^3)$ (see section 4.5.1) and the classification method $T^{class}_{max}(n)$ is $O(\log n)$ (see section 4.5.2). Given that $(T^{recall}_{max} + T^{class}_{max})(n)$ is $O(max(n^3, \log n)) \equiv O(n^3)$, the proposed learning algorithm has a cubic time complexity n^3.

The following sections describe the two methods of learning in detail.

4.5.1 The Recall Method

This section shows the way an *Experience* is selected from the agent's memory based on its relationship with the previous *Experience* and on the similarity between its conditions and the CURRENTCONDITIONS. The section also shows how the recall method calculates the RELIABILITY of the proposed solutions.

The recall method is inspired in sequencing, which is one of the most used methods in the surveyed approaches for learning by observation (see section 2.2.2). The sequencing approach relies on the sequence of actions executed by the expert, while it is performing a task, to tell the agent the sequence of actions it should execute to reach a specific goal.

Sequencing usually only uses the conditions to determine where to start following the sequence of actions. However, in the proposed approach, the current conditions determine which alternative paths to follow in the sequence. For example, an agent can decide which action to execute in a sequence depending on the conditions holding for those actions, as in a decision tree. The recall method and was totally developed for the proposed approach following this concept. The information stored in the agent's memory is regarded as a decision tree containing different approaches, provided by different experts, for accomplishing several tasks (see section 4.4).

Algorithm 5 Obtaining the REFERENCEEXPERIENCE $refEx$ from the agent's memory

Require: $cond$ to be of the same size as conditions in memory
Require: the agent to be in the execution state

```
function DISCOVERREFERENCEEXPERIENCE(cond,act):refEx
    similarity ← 0
    for all exp ← experiences in memory do
        if actions from exp are identical to act then
            if conditions from exp are identical to cond then
                return exp                                   ▷ found identical to reference
            else
                val ← similarityBetween(conditions in exp,cond)
                if val larger than similarity then
                    refEx ← exp
                    similarity ← val
                end if
            end if
        end if
    end for
    return refEx                                   ▷ returns the most similar to reference
end function
```

To propose actions, the method requires a reference to an *Experience* in the agent's memory,

called the REFERENCEEXPERIENCE, which indicates where to start following the connections in the tree. When the agent is in the learning state, the REFERENCEEXPERIENCE is the experience, stored in the agent's memory, concerning the snapshot that was observed before (see section 4.4). When the agent is in the execution state the REFERENCEEXPERIENCE represents the last sequence of actions executed, *act*, and the conditions holding, *cond*, when those actions were selected by the agent. Algorithm 5 shows the way the REFERENCEEXPERIENCE is discovered in the agent's memory when in the execution state.

The process in algorithm 5 shows that, when in the execution state, the REFERENCEEXPERIENCE is the experience in memory whose action is the same as the last action executed by the agent and whose conditions are the same as the conditions holding for the last executed action. Given that the agent might not be facing the same conditions as the expert, it is possible that the conditions holding for the last executed action are not found in the agent's memory. In these cases, the REFERENCEEXPERIENCE is the one with the same action who shares the largest number of similar conditions as calculated by the process shown in algorithm 6.

Algorithm 6 Calculating the similarity between two conditions, $condA$ and $condB$

```
Require: condA and condB to have the same size
  function SIMILARITYBETWEEN(condA,condB)
      sum ← 0
      size ← the size of condA
      for all cA ← condition in condA do                ▷ compare the conditions
          for all Cb ← condition in condB do
              if cA is identical to cB then
                  add 1 to sum
                  BREAK inner cycle
              end if
          end for
      end for
      return sum/size
  end function
```

After discovering the REFERENCEEXPERIENCE in the agent's memory, its tree subset (the set of tree branches) is retrieved. Figure 4.9 shows that the tree subset represents the choices of the *Experiences* that follow the REFERENCEEXPERIENCE. The sequence of action instances contained in those *Experiences* (displayed with a grey background in figure 4.9) represents the possible solutions (the possible sequences of actions) to be proposed by the recall method.

The RELIABILITY of a solution *sol* proposed by the recall method depends on two factors. The first one is the similarity between the conditions holding for the actions (in the experience)

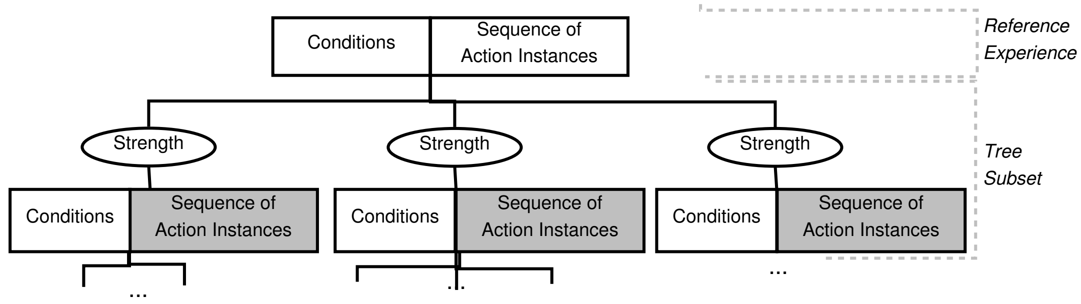


Figure 4.9: The action possibilities to be proposed by the recall method

condSol and the current conditions, which is calculated in algorithm 6. The second one is the strength of the connection to the REFERENCEEXPERIENCE *str*. Algorithm 7 shows the way the reliability is calculated for a proposed solution.

Algorithm 7 Calculating the RELIABILITY of a proposed solution sol by the recall method

Require: all conditions in the experiences in memory have the same size
  function RELIABILITYINRECALL($condSol, currentConditions, str$)
    $maxStr \leftarrow$ the maximum value for the strength in the tree set
    $sim \leftarrow$ similarityBetween($condSol$,$currentConditions$)
    if sim = 1 **then** ▷ conditions holding for sol are identical to $currentConditions$
      return sim x 0.99 + (str / $maxStr$) x 0.01 ▷ $strength$ has a minimal impact
    else ▷ conditions holding for sol are not identical to $currentConditions$
      return sim x 0.5 + (str / $maxStr$) x 0.5
    end if
  end function

As algorithm 7 shows, the highest reliability is assigned to the solution whose conditions are identical to the current conditions. When this happens, the strength of the connection to the REFERENCEEXPERIENCE has a minimal impact (only 1%) to prevent the algorithm from assigning a higher reliability to a solution whose conditions are not identical. In all other cases, both the similarity between the conditions and the strength of the connection have a similar impact because it is the simplest way of giving the same relevance for these factors.

The strength of the connection to the REFERENCEEXPERIENCE cannot be ignored even when the conditions are identical because there might be other solutions whose conditions are also identical to the CURRENTCONDITIONS. When this happens, the higher reliability is given to the solution with the highest strength because it is the one that was observed more often.

Considering a worst-case scenario, the time required to obtain the reference experience with n elements is $T_{max}^{refExp}(n) = n^3 + n$ and the time required to calculate the reliability is $T_{max}^{reliability}(n) = n^2 + 2$. The total time for the recall method is therefore $T_{max}^{recall}(n) =$

$T_{max}^{refExp}(n) + T_{max}^{reliability}(n) = n^3 + n^2 + n + 2$. For simplification purposes, the elementary unit of measurement regards only the operations on the elements of an *Experience*. The time expended by an elementary unit is considered to be greater or equal than the time taken when performing any of the actions it represents. Using the big-O notation it is possible to determine that $T_{max}^{recall}(n)$ is $O(n^3)$ and therefore the recall method has a cubic time complexity n^3.

Besides the recall method, the proposed learning algorithm provides another method for proposing actions, the classification method, which is explained in the following section.

4.5.2 The Classification Method

This section explains how the classification method was developed, what kinds of machine learning algorithms were assessed and how the agent's memory is used to train the machine learning algorithms. The section also shows how the classification method calculates the RELIABILITY of the solutions it proposes.

One of the tasks of this research focused on determining if a supervised machine learning approach could be used for interpreting the knowledge obtained from the observed actions, because learning by observation was often regarded as a subset of supervised learning. The classification method of proposing actions emerged from an analysis of the application of several known supervised machine learning algorithms presented in appendix C.

The analysis of the supervised learning algorithms concluded that the most suited algorithm for learning by observation is an implementation of the KStar algorithm provided by the WEKA (Waikato Environment for Knowledge Analysis) software workbench (Hall *et al.*, 2009). The classification method is an adaptation of the KStar algorithm to the proposed approach that enables the algorithm to propose representations of agent actions. The algorithm was also modified to allow the EXPERIENCES in the agent's memory to be regarded as positive examples from which the proposed actions are deduced. The categorization capabilities were also enhanced to allow the data-types of the conditions to define the differences between the classes.

Unlike the recall method, the classification method does not require a reference to an experience in the agent's memory to propose actions. It calculates the distances between the conditions to find the experiences whose conditions are closer to the current conditions. The KStar algorithm uses an entropy based function to measure the distance between the conditions (Cleary & Trigg, 1995). The algorithm was improved to allow the condition data-types to define the similarities between the conditions. For example, the *Text* data-type can define that the

instance *"hello"* is identical to the instance *"HELLO"*. The data-type can also define that the instance *"hello"* is closer to the instance *"hi"* than the instance *"goodbye"*.

According to Clearly and Trigg (Cleary & Trigg, 1995), the KStar function can be calculated as $K^*(b|a) = -\log P^*(b|a)$, where P^* is the probability function representing all paths from a to b. Using the big-O notation it is possible to determine that $K^*(n)$ is $O(\log n)$. Since the improvements made to the KStar algorithm had minimal impact on the time complexity, the classification method has a logarithmic time complexity $\log n$.

The RELIABILITY of the actions proposed by the classification method is directly associated with how much the conditions holding for that action are close to the current conditions. Once again, the modified entropy function is used to measure this distance. For example, if the conditions holding for a proposed action are identical to the current conditions the RELIABILITY of that action is 1, that is, the action is fully reliable from the standpoint of the classification method.

In addition to the learning algorithm, another important aspect of the proposed approach is the constant evaluation of the agent's knowledge, which is explained in the following section.

4.6 Evaluation of the Agent's Progress

This section shows the way the agent's evaluation operates and how it affects the transition between the two states of the learning process, the learning and the execution state. It describes the way the agent's confidence is updated and how it is related with the agent's ability to propose appropriate actions. The section also shows that evaluation goes beyond controlling the confidence value and the transition between the learning and execution states, it is also determinant for ensuring that the apprentice does not execute incorrect sequences of actions.

The evaluation is a transversal process that covers both the learning and the execution state. The main purpose of evaluation is to ensure that the agent's knowledge is appropriate for mastering a task, which influences the agent's internal confidence. The agent's internal confidence expresses the successes and failures in proposing the appropriate actions for the current conditions. Depending on the value of the internal confidence the agent may be in the learning state or in the execution state of the learning process. Two configurable thresholds, the UPPERCONFIDENCETHRESHOLD and the LOWERCONFIDENCETHRESHOLD determine the values at which the agent changes to the learning or to the execution state. Section 5.2.3 shows how the

values selected for these thresholds influence the agent's performance.

When the agent's confidence goes over the UPPERCONFIDENCETHRESHOLD the agent is confident enough on its knowledge and switches to the execution state where it executes the actions it proposes (see section 4.5) for the conditions provided by its sensors and by the relevant aspects of its internal state (the visible attributes). When the confidence goes under the LOWERCONFIDENCETHRESHOLD the agent stops being confident on its knowledge and switches to the learning state where it acquires more knowledge by observing experts (see figure 4.2). The value of the internal confidence is affected by the agent's capacity to propose the appropriate actions both when it is observing (in the learning state) and when it is preparing to execute them (in the execution state).

In the learning state, the agent is tested for its ability to propose actions for the conditions in the observed snapshots. The agent's confidence increases when the best solution it proposes (the one with the highest FINALRELIABILITY as explained in section 4.5) contains the same actions as the ones observed in the snapshot, otherwise the confidence decreases. The amount with which the confidence increases or decreases depends on the RELIABILITY of the proposed solution. For example, if A is the action in the snapshot, B is the action from the best proposed solution (with a RELIABILITY of 0.8) and $A \neq B$, the agent's confidence decreases 0.8.

Using the RELIABILITY of the best proposed solution as a factor for increasing or decreasing the agent's confidence is the simplest way of including the confidence the learning methods have on the solutions they propose (see section 4.5) in the calculation of the agent's internal confidence. The way the internal confidence is updated gives more importance to the solutions with a high RELIABILITY, that is, when the learning methods have a high confidence on the actions they propose. In these cases, if the solution is not correct the penalization in the internal confidence should be larger than when the methods have low confidence on the actions they propose because it means the agent learnt something wrong.

In the execution state, the agent is tested for its ability to execute the correct actions for the current conditions, which reflect the agent's perception from its sensors and the important aspects of its internal state. As section 4.5 shows, the agent selects the solution with the highest FINALRELIABILITY, from those proposed by the recall and classification methods, to be executed. A simple monitoring detects if there was any problem that prevented the correct execution of the actions in that solution. Whenever a problem is detected the agent's confidence decreases. Once again the amount with which the confidence decreases depends on the

RELIABILITY of the selected solution.

The simple monitoring of the executed actions has some limitations because even though there are no problems when executing the actions it is not possible to ensure that the selected solution was the most appropriate for the current conditions. To overcome this problem, the proposed approach extends the concept of learning by observation by allowing the evaluation to receive external feedback on the executed actions from specialized experts, the teachers, or from other evaluators that are specific to the application domain. This possibility for evaluation approximates the approach to the paradigm of learning by teaching.

The teachers are a special kind of experts that know how to evaluate the task executed by the agent. They observe the actions executed by the agent and determine if those actions were appropriate for the current conditions. They provide evaluation with a positive or negative feedback on the executed actions. The evaluation can also receive basic feedback (positive or negative) from software components that are specialized in evaluating certain aspects of agent tasks when teachers are not available. The components know which actions are appropriate for the current conditions and use this knowledge to determine if the agent executed the same actions.

The external feedback influences the agent's confidence in the same way as the monitoring of the execution of the actions. If the feedback is positive, meaning the agent was able to execute the appropriate actions, the confidence increases. If the feedback is negative, meaning the agent has not executed the appropriate actions, the confidence decreases. Once again, the amount with which the confidence increases or decreases depends on the RELIABILITY of the proposed actions.

In addition to decreasing the agent's confidence, which may cause the agent to return to the learning state, the negative feedback provides useful information on what the agent should not do. It represents the actions that cannot be executed when facing certain conditions - a negative training example. When it is possible to use external feedback, the evaluation collects the negative training examples and uses them to prevent the agent from executing the actions in those examples when facing the same conditions. Each time the agent selects the actions to execute from the solutions provided by the learning methods (see section 4.5), the selected actions are validated before the agent executes them, as shown in algorithm 8.

As algorithm 8 shows, the validation consists of comparing the conditions in the negative examples with the current conditions. If identical conditions are found in a negative example

Algorithm 8 Validating the sequence of actions $seqAct$

```
function VALIDATE(seqAct)                     ▷ allows or blocks the execution of seqAct
    pCond ← the current conditions
    nteList ← list of negative training examples
    for all nte ← example in nteList do       ▷ go through all negative examples
        cond ← conditions in nte
        if cond are identical to pCond then
            act ← the actions in nte
            if act is identical to seqAct then
                return blockActions           ▷ blocks the execution of the actions
            end if
        end if
    end for
    return allowActions                       ▷ allows the execution of the actions
end function
```

and the actions of that negative example are identical to the actions of the selected solution, the agent is prevented from executing those actions, which forces the actions from the next best solution to be selected and validated once again. This happens until the selected actions are approved and executed, or until there are no more solutions, which causes the agent to execute no actions.

In addition to the external feedback that changes the agent's confidence, the agent can be forced to switch to the learning state independently of its confidence. This additional possibility of switching to the learning state happens when the agent is faced with a certain amount of unfamiliar conditions, that is, conditions not found on the observed snapshots. The amount of unfamiliar conditions is controlled by a configurable threshold, the AMOUNTUNFAMILIAR-CONDITIONS, whose value depends of the application domain. As explained in section 3.2.2, the conditions consist of the information provided by the agent sensors and visible attributes. The conditions are familiar when the agent has observed an expert facing the same conditions and therefore knows exactly the correct actions to propose.

If the conditions are not familiar, the agent has never observed an expert facing them which probably means that it does not know what actions to propose. Under these circumstances, the amount of unfamiliar conditions rises only when the evaluation determines that the executed action was inappropriate. To turn this additional possibility to switch to the learning state in a measure of last resort, the amount of unfamiliar conditions resets each time the agent faces a familiar condition. Therefore, this amount represents the number of consecutive times the agent faces unfamiliar conditions.

After switching to the learning state because of the amount of unfamiliar conditions, the agent's confidence is decreased to a configurable reference value, the UNFAMILARCONFIDENCEREFERENCE, which has to be lower than the LOWERCONFIDENCETHRESHOLD to allow the agent to remain in the learning state for a while. The lower the value of UNFAMILARCONFIDENCEREFERENCE in relation to the LOWERCONFIDENCETHRESHOLD the longer the agent takes to change back to the execution state. This is the simplest solution to keep the agent in the learning state for some time.

Other solutions could take the unfamiliar conditions into account. For example, the agent can remain in the learning state until it finds an expert that is facing those unfamiliar conditions. However, the solution requires a complex timeout mechanism to prevent the agent from remain in the learning state for a long time if it does not find an expert facing the unfamiliar conditions. A similar behaviour can be easily achieved by reducing the confidence and letting the internal evaluation decide when to return to the execution state by testing the agent's knowledge while it is observing.

The role played by evaluation, namely making the agent return to the learning state, contributes to overcome one of the major problems faced by learning by observation, the fact that the agents are limited to performing only the actions they have observed. Forcing the agent to return to the learning state (and thus to observe other experts) gives the agent an opportunity to increase its knowledge since other experts may have different experiences that might provide the agent with new knowledge. This is validated in the following section where the approach is tested in different scenarios were experts have different experiences.

Chapter 5

Application Scenarios

This chapter describes three scenarios in which the proposed approach was implemented. It also presents the tests performed on the proposed software image (see chapter 3). The chapter ends with the validation of the evaluation criteria defined in section 1.3. Since the scenarios are software implementations, the observation noise when acquiring the snapshots or the current conditions makes no sense and therefore is not considered in the experiments.

The tests on the software image determine the time to create, compare and observe the software images of different agents (see section 5.1). In the first scenario, the agent learns how to manipulate their virtual hand to display numbers using sign language (see section 5.2). In the second scenario, the agent learns how to calculate mathematical expressions using the numbers and operators acquired from the environment (see section 5.3). In the third scenario, the agent learns how to reach the top of a simulated mountain by going backwards and forwards to gain momentum (see section 5.4).

The tests on the software image determine how the complexity of an agent affects the creation, comparison and observation of software images. The agent complexity is measured by the number of parts, the number of atomic elements in those parts and the depth of sub-parts. The tests also determine how accurate is the software image comparison process when comparing similar software images.

The first scenario was designed to be appropriate for learning by observation because the majority of the agent actions does not affect the state of the environment. It was specially conceived to test the abilities of the proposed approach. It shows the way the configurable confidence thresholds influence the agent's capacity to learn, the way the proposed meta-ontology (see section 3.3) enhances software image comparison and also the influence of the two methods

of proposing actions, the recall and classification methods, when the expert and the apprentice agent face the same conditions and when they face different conditions.

The second and third scenarios present a direct comparison between the proposed approach and implementations of the KStar, multi-layer perceptron back-propagation (MLP) and reinforcement learning algorithms. The statistical relevance of the data collected for the comparisons is ensured by the student's t-test.

The second scenario (see section 5.3) compares the proposed approach with implementations of the KStar and of the multi-layer perceptron back-propagation algorithms that only learn from the effects of the actions. The scenario presents a situation where not all the effects of the agent actions are visible in the environment.

The third scenario (see section 5.4) compares the proposed approach with a reinforcement learning approach in a typical reinforcement learning experiment, the mountain car experiment from Sutton and Barto (Sutton & Barto, 1998). In this scenario all the agent actions are visible in the environment.

The scenarios are software implementations where all developed agents have a software image. When apprentice software agents observe the experts they are effectively acquiring snapshots from the expert's software image. As a simplification, the software images of the apprentice agents have the same constituents as the software images of the experts they observe. With the exception of one of the settings of the first scenario (see section 5.2.4), in all scenarios the apprentice agents do not receive external feedback from teachers or other evaluators (see section 4.6) so that the results reflect a pure learning by observation approach.

The process of discovering and identifying a similar expert is an important step for the proposed approach. However, for the purpose of focusing the results on the benefits of learning by observation, this process is only considered in the tests of the proposed software image (see section 5.1) and in the first scenario (see section 5.2.5). In the remaining scenarios it is implicit that the apprentice agent uses the software image for discovering, identifying and collecting the information that is necessary for observing the experts.

The following sections describe the scenarios and the results of the experiments. To provide the learning by observation agent with a broader set of experiences, the experiments use more than one expert and each expert experiences the scenario in different ways, that is, they receive different information from the environment through their sensors. On the other hand, only one apprentice agent is used. This simplification ensures that there is no risk for the tested

apprentice agent to observe other apprentices, which may mislead it with incorrect actions, instead of observing the experts.

The unit of time used for the experimental results is the simulation step. A simulation step represents the time slot where the participant agents take a decision. For learning by observation agents, a simulation step can either represent an observation (the acquisition of a snapshot from the expert) and the subsequent learning (when the agent is in the learning state), or updating the current conditions, proposing and executing the best suited actions for those conditions (when the agent is in the execution state) (see chapter 4).

For the KStar and the multi-layer perceptron back-propagation learning agents a simulation step can represent collecting a training example when the agents are in the learning state or updating the current conditions, selecting the actions for those conditions (training the algorithm with the collected training examples and proposing actions) and executing the actions when in the execution state.

For reinforcement learning agents a simulation step represents updating the current conditions, selecting an action for those conditions (from the action-state pairs stored in their memories) executing the action and interpreting the rewards (updating the action-state pairs).

For expert agents, a simulation step represents updating the current conditions and using their control knowledge to select and execute the best suited actions.

5.1 Tests on the Software Image

This section presents the tests performed on the proposed software image described in chapter 3. The first part of the test (section 5.1.1) provides a measure of the agent's complexity and shows how this complexity affects the time to create the software image and to discover and observe an identical agent. The complexity of an agent depends on the number of parts, the number of atomic elements in those parts and the depth of sub-parts.

The second part of the test (section 5.1.2) shows the capabilities of the software image comparison process. It determines the accuracy of the software image comparison process and the time it takes to compare different software images.

5.1.1 Creating and Observing the Software Image

This section shows how the agent complexity affects the time to create the software image. It shows the effect of the process of creating the software image in the time it takes to initialize the agent. It also shows how the agent complexity affects the time it takes to discover and observe an agent.

The agent's complexity is measured by three aspects of the agent constituents: the number of parts, the number of elements in each part and the depth (agent parts inside of agent parts). The test uses six agents with different arrangements of parts, elements and depth to determine which of these aspects has more influence in the time to create and observe the software image. Table 5.1 shows the arrangement of the parts, elements and depth in the tested agents.

	Agent1	Agent2	Agent3	Agent4	Agent5	Agent6
No. of parts	1	1	1	20	1	1
No. of sensors and visible attributes in part	2	40	40	2	4	2
No. of actuators in part	1	20	1	1	2	1
No. of actions in actuator	1	1	20	1	1	1
Depth	0	0	0	0	1	20

Table 5.1: Description of the agents used in the test

The arrangement in table 5.1 determines the influence of several parameters of the agent architecture in the agent's complexity. It determines how an increase in a factor of twenty on the number of agent parts, the number of elements in the parts or the depth influences the results of the tests. It also determines the influence of the distribution of the agent actions in the actuators.

As section 3.4.1 shows, the software image is created when the agent is initialized. The tested agents, described in table 5.1, are initialized to determine the time it takes to create the software image. In addition, the test also determines the proportion of time spent by the process of creating the software image in the time it takes to initialize the agent. Table 5.2 presents both the average values and the standard deviation from running the test 100 times to encompass the time variations.

		Agent1	Agent2	Agent3	Agent4	Agent5	Agent6
Time to create the software image (sec.)	Avg	0.1	0.37	0.25	0.22	0.17	0.29
	StDev	0.03	0.1	0.08	0.07	0.07	0.07
Proportion of time in agent initialization	Avg	95.4%	94.9%	92.6%	96%	87.2%	70.2%
	StDev	3.1 pp	1.5 pp	1.7 pp	2.3 pp	1.4 pp	2.6 pp

Table 5.2: Results for the test on creating the software image

The results in table 5.2 show that it takes less time to create the software image when the number of parts, elements in the part and depth are small (Agent1). The software image creation process is largely affected by the increase of the number of elements in the agent parts and the depth (Agent2 and Agent6). The software image also takes less time to create when all the actions are in the same actuator (Agent3) instead of being one in each actuator (Agent2).

The results also show that the software image creation process is responsible for almost all the time required to initialize the agent (more than 90%). This proportion is lower (about 70%) only when the agent has a large depth (Agent6). Even though this is one of the situations that requires more time to create the software image the agent takes even longer to initialize.

A second test on the software image determines the effect of the agent complexity in the time it takes to discover and to observe an expert. The test also determines the proportion of time spent by the software image comparison process in the time it takes to discover the agent. The test uses six apprentice agents (Ap1 to Ap6) that are identical to the agents described in table 5.1. The apprentices have to discover their identical expert from the six agents of the previous experiment and observe the discovered expert, that is, collect a snapshot from the expert's dynamic image. Table 5.3 presents both the average values and the standard deviation from running this test 100 times to encompass the time variations.

		Ap1	Ap2	Ap3	Ap4	Ap5	Ap6
Time to discover expert agent (sec.)	Avg	0.38	41.35	157.35	0.39	29.9	38.2
	StDev	0.06	2.01	13.9	0.05	0.29	1.56
Proportion of time spent comparing expert	Avg	95.2%	86.13%	89.6%	94.4%	93.2%	95.4%
	StDev	1.2 pp	2.1 pp	1.3 pp	1.4 pp	1.1 pp	2.2 pp
Time to acquire snapshot from expert (sec.)	Avg	0.01	0.012	0.014	0.01	0.014	0.01
	StDev	0.003	0.002	0.004	0.002	0.004	0.002

Table 5.3: Results for the test on discovering experts and observing the software image

The results in table 5.3 show that the software image comparison process is responsible for about 90% of the time spent to discover the expert. The results indicate that the discovery process is largely affected by the complexity of the apprentice because of its impact in the com-

parison process (see section 5.1.2). It takes less time to discover experts when the apprentice has a small number of parts, elements in the part and depth (Ap1). The results also show that the discovery of expert agents is largely affected by the increase of the number of elements in the agent parts, especially when the actuators have a large number of actions (Ap3).

Table 5.3 also shows that observation is not affected by the agent complexity. The time spent acquiring a snapshot from the expert's dynamic image is the same for all tested apprentices. The time fluctuations in the results can be disregarded because they are inside the standard deviation.

5.1.2 Comparing Software Images

This section presents two tests on the software image comparison process described in section 3.4.3. The first one shows the capabilities of the software image comparison process when comparing different software images. The second one shows the performance of the software image comparison process in terms of the time it takes to compare different kinds of software images.

The software image comparison process determines if two agents have the same constituents and capabilities, that is, if two static images have the same sensors, actions and visible attributes arranged in the same way (see section 3.4.3). The comparison process is based on tree comparison because of the way the static image is organized (see section 3.2.1).

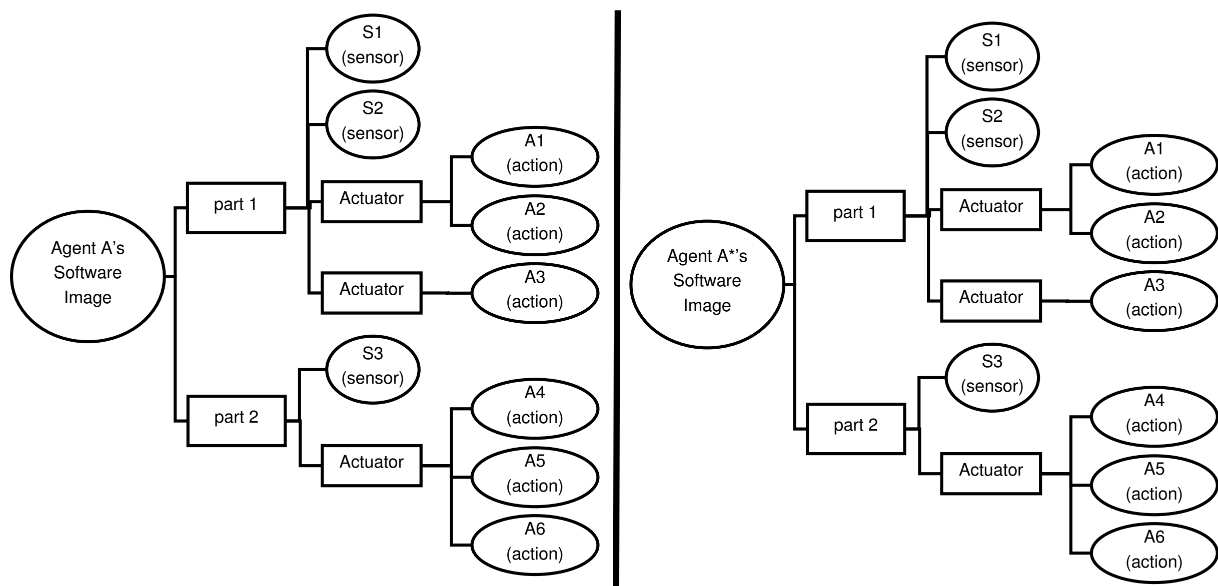


Figure 5.1: The static images of agent A and $A*$

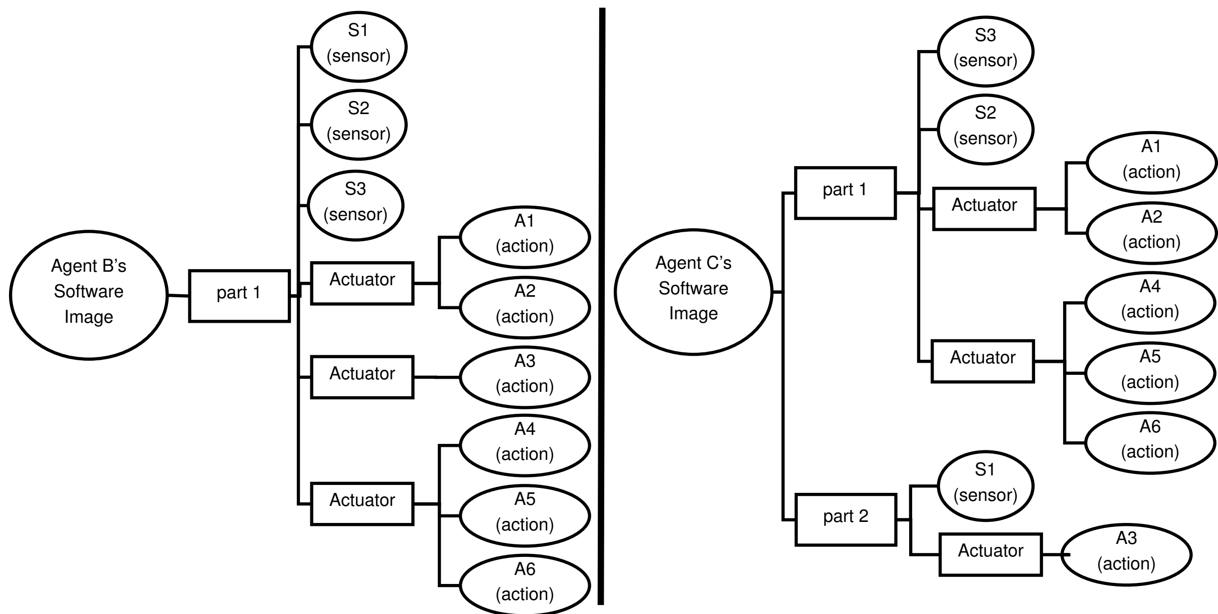


Figure 5.2: The static images of agent *B* and *C*

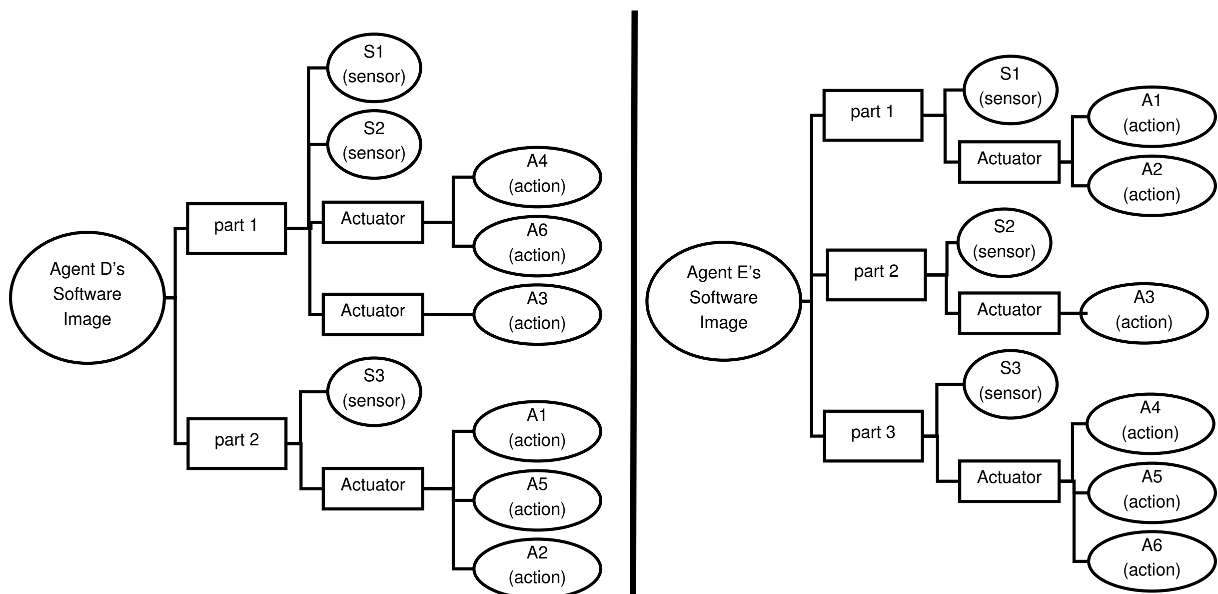


Figure 5.3: The static images of agent *D* and *E*

The accuracy of the comparison process is determined by comparing static images that have small differences between them. The test uses the static images of six agents, shown in figures 5.1, 5.2 and 5.3. It consists of comparing the static image of agent *A* with the static images of the remaining agents *A**,*B*,*C*, *D* and *E*. All static images have the same elements but the static image of agent *A** is the only one where the elements are organized in the same way as the static image of agent *A*. Thus, the static image of agent *A** is the only one that is identical to *A*.

The test also determines the time it takes to compare the static image of agent A with the remaining static images. Results show that the comparison process correctly determines the identical and non-identical static images. Figure 5.4 shows the results from repeating the test 5000 times to encompass the time variations. It presents the average time taken to create the software images of each agent and the time it takes to compare them with the static image of agent *A*.

Figure 5.4: Results for the first software image comparison test

Figure 5.4 shows that comparing software images spends less time than creating them even though both processes are similar. As explained in section 3.4.1, the software image creation process uses code introspection to discover the elements of the software image, which takes more time to complete the process. The fact that creating the software image takes more time is not disadvantageous because software images are created once but can be compared many times.

Figure 5.4 also shows that comparing identical images (*A* and *A**) takes more time than comparing images that are not similar. This situation requires the largest number of iterations because it is necessary to compare all elements with each other to prove that both images are identical.

The second test determines how the complexity of the agent affects the performance of the comparison process in terms of the time and memory required for comparing two identical static

images. The test shows how the performance is affected when changing the number of agent parts, the number of visible elements (sensors, actuators, visual attributes and internal agent parts) and the depth of the related agent parts.

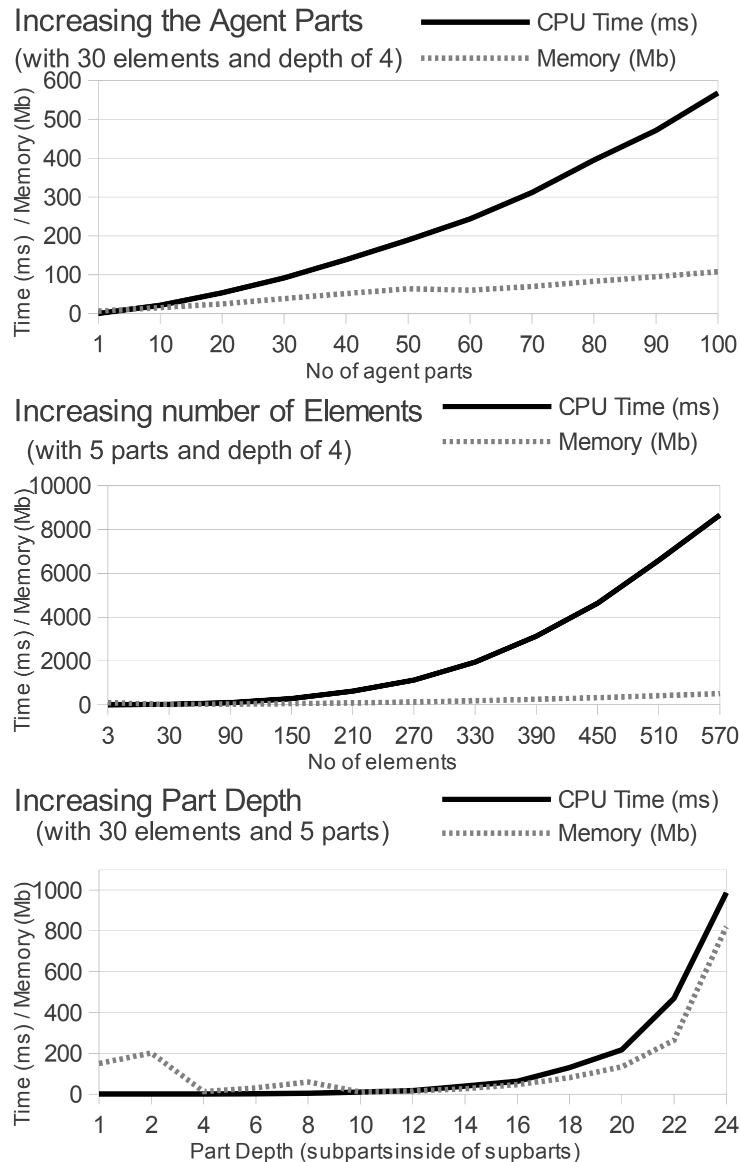


Figure 5.5: Results for the second software image comparison test

Figure 5.5 shows the results from repeating the second test 5000 times to encompass the time variations. To provide a large number of different possibilities, a new pair of identical software images was randomly generated in each comparison. The random generation ensures minimal deviation of the non-varying parameters (depth, number of elements or number of agent parts). The random generation also used a set of parameters that ensured minimal deviation for the tested depth, number of elements and number of agent parts.

Figure 5.5 displays the results of the test in three graphics. The *x* axis represents the test progression (the increments of the number of parts, elements and depth). The *y* axis represents the average time taken, in milliseconds, and the memory required, in megabytes, by the comparison process.

The first graphic shows the effect of increasing the number of agent parts while maintaining the same number of elements (thirty elements) and the same depth (four levels of internal parts). The second graphic shows the effect of increasing the number of elements while maintaining the same number of agent parts (five agent parts) and the same depth (four levels). The third graphic shows the effect of increasing the depth while maintaining the same number of elements (thirty elements) and the same number of agent parts (five agent parts).

The graphics in figure 5.5 show that the time required for the comparison process increases exponentially when any of the three tested aspects increase. The exponential growth is more pronounced when increasing the number of elements and the depth. Increasing the depth is also more significant when considering the amount of memory required for comparison.

The results show that building agents with a large number of small parts (with a small number of elements and depth) is a better choice than building agents with few parts that have lots of elements or a large depth. These results support the design choice of decomposing a problem into smaller problems, each one solved by a simpler agent part.

5.2 The Virtual Hand Scenario

This section describes the first scenario in which the proposed approach was implemented and shows the results of simulating this scenario in different settings. The scenario simulates an agent with a virtual hand that displays numbers using sign language. The numbers are provided to the agent by a software number generator which provides numbers between one and five.

Each agent is associated to a single number generator and has no knowledge of the other number generators associated to the other agents that participate in the scenario. Each time a new number is provided to the agent, the virtual hand is changed to display that number by means of sign language. The virtual hand is used only for communicating with users through a graphical display and it is not accessible to other agents, except by using the agent software image.

The only way software agents can obtain information on the state of the agent's virtual hand

is through a visible attribute in the software image (see section 3.2), the *hand* visible attribute. The *hand* visible attribute represents the state of the visible hand as an object with five attributes each representing one finger. Each attribute can be in one of two states, UP or DOWN. Figure 5.6 shows a representation of the type of data provided by the *hand* visible attribute.

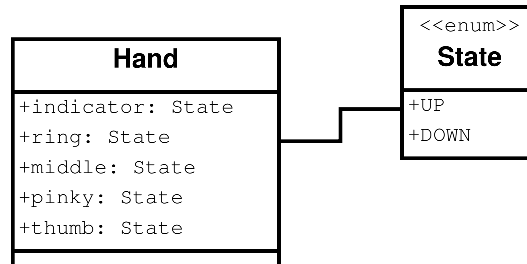


Figure 5.6: The data-type of the *hand* visible attribute

The expert agent designed for this scenario has a specialized part that holds the knowledge on the most efficient way of changing the virtual hand so that it represents the number provided by the number generator. For example, when the agent perceives the number one, if the virtual hand is showing the number two (the index and middle fingers are UP) it is only necessary to change the state of the middle finger to DOWN, whereas if the hand was showing the number four, it would be necessary to change the state of the middle, ring and pinky fingers to DOWN. The part perceives the numbers from the number generator through a specialized sensor. It has an actuator with five actions, one for each finger, that change the state of that finger.

To increase the complexity of the scenario, the number generators need to be reset from time to time or else they will stop generating new numbers. The number sources can either be in the ACTIVE state or in the INACTIVE state. The source stops providing numbers when it is INACTIVE. Resetting the source leaves it in the ACTIVE state. Because of this, the expert agent has another specialized part which holds the knowledge on when and how to reset the random number generator. The state of the number generator is perceived by this specialized part through a sensor which indicates the state of the generator. The agent part has one actuator with a single action that resets the source.

The apprentice agent developed for this scenario has to learn how to master these two tasks, manipulating the agent's virtual hand to display the perceived numbers and resetting the source, by observing experts with the same constituents and capabilities. Like the experts, apprentice agents have two parts. One of them specializes in manipulating the virtual hand and has one sensor that perceives the numbers provided by the number generator, one visible attribute that displays the state of the virtual hand and one actuator with five actions that change the state of

each finger.

The other part of the apprentice agent specializes in managing the source that provides the numbers and has one sensor that perceives the state of the number source and one actuator with a single action that resets the source. Given the description of the agent sensors and visible attributes, the conditions for this scenario (see section 3.2.2) consist of the number provided by the agent source, the state of the virtual hand and the state of the number source. Appendix D shows the distribution of the elements that participate in the scenario.

The scenario was experimented in four different settings that depend on the sequence of numbers provided by the number generators and on the visibility of the *hand* visible attribute, as shown in table 5.4. To control the results of the experiment, the numbers provided by the generators follow a pre-determined sequence. The number generators provide the numbers following that sequence and when the end of the sequence is reached the source needs to be reset by the agent. Resetting the number generator makes it provide the numbers following the same sequence.

Setting	Sequences in number generator	*hand* visible attribute
exp1	same for experts and apprentice	is visible
exp2	different for experts and apprentice	is visible
exp3	same for experts and apprentice	not visible
exp4	different for experts and apprentice	not visible

Table 5.4: The settings for the virtual hand scenario

Table 5.4 shows that on the first (*exp1*) and third settings (*exp3*), the number generators of both apprentice and expert agents provide the same sequence of numbers. In this setting both the expert and the apprentice agent face the same conditions in the same sequence, which provides a good testing ground for the recall method (see section 4.5.1). On the second (*exp2*) and fourth setting (*exp4*), the number generators have different sequences of numbers with different sizes, that is, the number generators need to be reset at different times. The expert and the apprentice agent face different conditions which provides a good testing ground for the classification method of learning (see section 4.5.2).

The difference between the first two (*exp1* and *exp2*) and the last two settings (*exp3* and *exp4*) is the visibility of the *hand*. On *exp1* and *exp2* the *hand* is visible in the agent's software image whereas in *exp3* and *exp4*, the *hand* is not visible in the software image. Since the information on the state of the hand is not visible, it will not be in the conditions of the observed snapshots. Thereby, apprentices face a situation where important information is missing. With-

out this information the same set of conditions may lead to different actions since the decision to perform those actions also depends on the state of the *hand*.

A practical example of this situation is when the agent is observing an expert and the expert's number generator provides the number one. When this happens, the expert can either change the state of the index finger if all fingers are down or change the state of the middle finger if the index and middle fingers are up. If the information on the current state of the agent's hand is removed from the conditions in the observed snapshots, those conditions will only contain information on the number provided by the generator source. When the agent is applying the knowledge acquired from those snapshots, whenever the source provides the number one, it can either change the state of the index finger or change the state of the middle finger. Without the visible attribute to tell the agent what should be the state of its hand when proposing actions, the apprentice does not know which of the actions to execute.

As described in appendix C, this kind of situation is usually catastrophic for the majority of supervised learning algorithms, because for each set of conditions there can be more than one possible action. Therefore, the two last settings (*exp3* and *exp4*) determine how the approach to learning by observation copes with the lack of important information.

The virtual hand scenario is the background for several experiments with the proposed approach. The following sections show the results of these experiments in these four settings.

5.2.1 Results for the First Two Settings

This section presents the results of testing the scenario for the first two settings (see table 5.4) in a 4000 step experiment which was repeated 100 times to encompass the time variations that might exist in the experiments. The configurable parameters were set with the optimal values, 0.3 for the UPPERCONFIDENCETHRESHOLD, 0.05 for the LOWERCONFIDENCETHRESHOLD, 3 for the AMOUNTUNFAMILIARCONDITIONS and 0.04 for the UNFAMILARCONFIDENCEREFERENCE. Section 5.2.3 explains how these optimal values can be obtained.

Setting		Recall Weight	Classification Weight
exp1	avg	0.509	0.491
	stdev	0.008	0.008
exp2	avg	0.317	0.683
	stdev	0.022	0.022

Table 5.5: The average weights of the recall and classification methods on each setting

Since the two settings were prepared with the two learning methods in mind, it is important to look at the importance given by the agent to these methods on each setting. Table 5.5 shows the normalized values of the weights of the recall and classification methods of learning (see section 4.5) on each setting.

Table 5.5 shows that, as expected, the recall method has more influence on the proposed actions when the expert and the apprentice agent face the same conditions (*exp1*) than when they face different conditions (*exp2*). In the first setting, the weight of the recall method makes it more likely (more than 50% chance) for the actions proposed by this method to be executed, even though the actions proposed by the classification method can also be executed. In the second setting, the classification method has more influence on the proposed actions because it has the largest weight. The agent is not able to use the recall method very often to follow the same sequence of actions as the expert because it is facing conditions that are different from those observed on the expert (see section 4.5).

The overview of the results of the scenario experiment in *exp1* and *exp2* is presented in table 5.6. The table compares the time spent on the learning state and on the execution state, the number of actions the agent was able to execute, how many of those actions were appropriate and also the time of a simulation step when learning and when executing actions. The results in table 5.6 present both the average values and the standard deviation from running the experiment 100 times.

		Expert	Apprentice (exp1)	Apprentice (exp2)
Time spent learning (seconds)	average	-	2.09	177.5
	stdev	-	0.54	45.43
Time spent execution (seconds)	average	1.72	11.44	16.75
	stdev	0.46	1.76	4.34
Total actions executed	average	4000	3931	3588.7
	stdev	0	0	853.33
% of appropriate actions executed	average	100%	100%	93.41%
	stdev	0	0	10.15 p.p.
Step time in learning state (ms)	average	-	32.17	269.41
	stdev	-	10.5	153
Step time in execution state (ms)	average	0.43	2.9	4.68
	stdev	0.115	0.45	0.67

Table 5.6: Results of experimenting the first two settings of the virtual hand scenario

The results in table 5.6 show that when the expert and the apprentice agent face the same conditions (*exp1*), the apprentice spends less time in the learning state and is able to perform

more actions throughout the experiment than when facing different conditions (*exp2*). When the expert and the apprentice agent face different conditions (*exp2*), the apprentice spends more time learning because it needs to acquire more knowledge and requires this knowledge to come from different sources to improve the generalization in the classification method (see section 4.5.2). This leads the agent to spend more time discovering different experts to learn from and also causes the agent to return to the learning state more often. Section 5.1.1 shows how the time spent discovering an expert influences the time spent learning.

In addition, the high standard deviation values show that when the agent faces different conditions (*exp2*) the number of actions executed throughout the experiment, the time spent learning, the time spent executing actions and the simulation step time when learning differs considerably. This variation is directly associated with the randomness of the observed experts. Each time the experiment is run, the apprentice observes different subsets of the available experts in a different sequence, which changes the knowledge retained by the agent throughout the experiment and affects the number of actions executed and the time spent on learning and on executing actions.

Table 5.6 also shows that when facing the same conditions (*exp1*) all the actions executed by the agent were appropriate, whereas when facing different conditions (*exp2*) only approximately 93% of the executed actions were appropriate. This is an important indicator that the agent is more likely to return to the learning state, after starting to execute actions, when it is facing different conditions (*exp2*).

As for the average time spent in the simulation steps, table 5.6 shows that both in *exp1* and *exp2*, the simulation step is longer when agents are learning. This was expected since when the agent is in the learning state it has to perform various tasks such as discovering expert agents, comparing its software image with the software images of the discovered experts, acquiring the snapshots and storing its information, proposing actions for the conditions from the snapshots and evaluating those proposals (see chapter 4).

When executing actions, the simulation step of the apprentice agents is longer than the simulation step of the experts, which is understandable given that apprentice agents require more processing than the expert. Besides using two methods for proposing actions it is also necessary to take into account the influence of the agent's evaluation (see section 4.6). The step time while executing actions is also longer when the apprentice agent is facing different conditions (*exp2*).

Table 5.6 shows that, when compared with *exp1*, in *exp2* the agent executes fewer actions and spends more time executing them (an average of 3589 actions in *exp2* and 3931 actions in all trials in *exp1*). The same effect is observed in the time spent by the simulation step when learning. When facing different conditions (*exp2*) the agent needs to acquire more knowledge which increases the amount of information in the agent's memory. Since the recall and classification methods of learning need to process the information contained in memory, the larger the amount of information the longer it takes to process it. This effect is felt both when learning and when executing actions because the agent uses these methods in both cases.

A closer look at the experiment results for *exp1* and *exp2* is presented in figure 5.7, which shows the progress of the agent, in terms of the number of correct actions it has executed, throughout the experiment. The figure also shows the progress of the state of the learning process (see chapter 4) throughout the experiment. To provide a clearer presentation, the results are combined in groups of 100 simulation steps.

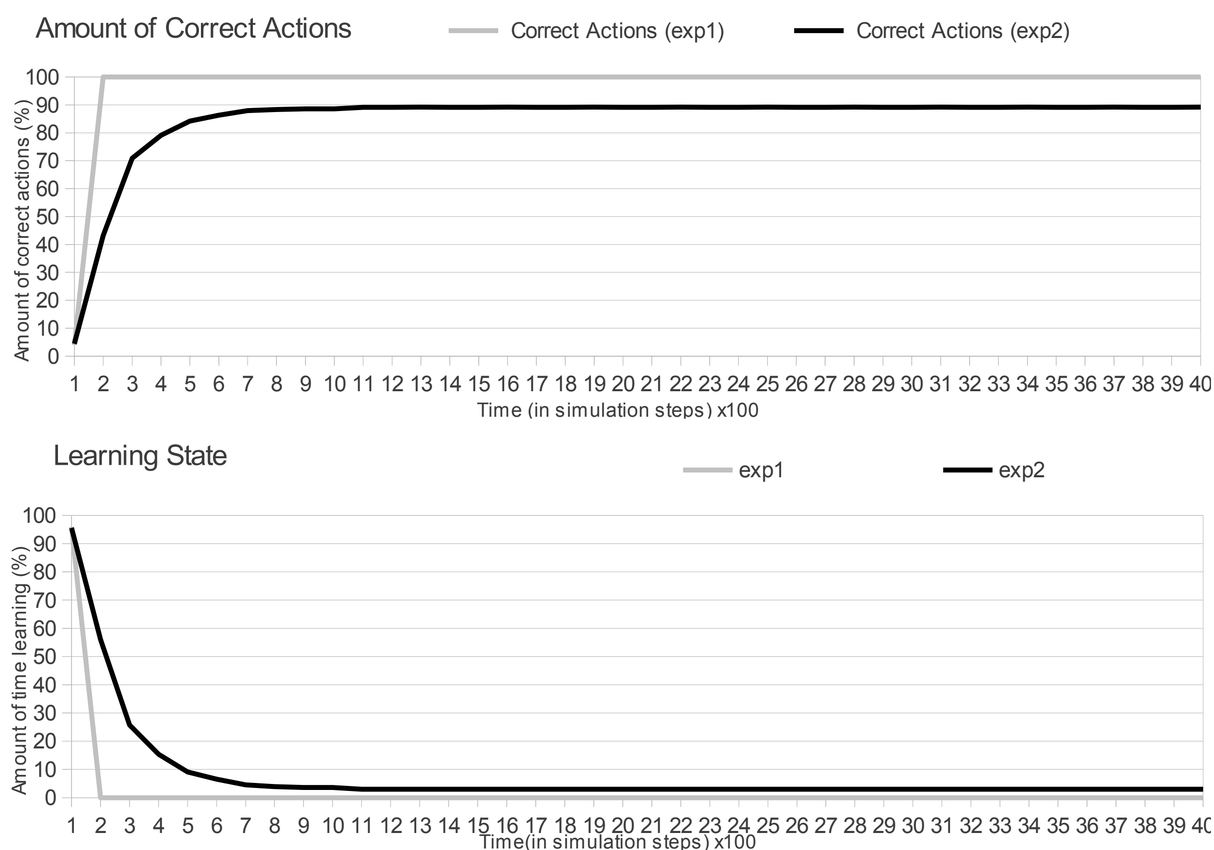


Figure 5.7: The progress of the agent in *exp1* and *exp2* throughout the experiment

Figure 5.7 shows that, when the expert and the apprentice agent face the same conditions (*exp1*) after a short learning period (of about 200 steps) all the actions executed by the apprentice

are correct. When the expert and the apprentice agent face different conditions (*exp2*), the learning period lasts longer than when they face the same conditions and only approximately 90% of the actions executed by the apprentice are correct.

The agent also experiences additional learning periods (of short duration) throughout the experiment. The additional learning periods are mainly caused by the evaluation activity mechanism that switches the agent to the learning state when facing conditions that were not observed (see section 4.6). Although the subsequent learning periods affect the time spent on learning they are of short duration. As figure 5.7 shows, the overall permanence in these subsequent learning periods does not exceed more than 3% of the total simulation steps.

The scenario was also experimented in a situation where the *hand* is not visible in the agent software image (*exp3* and *exp4* from table 5.4). The following section presents the results of simulating the scenario in these settings.

5.2.2 Results for the Last Two Settings

As in the first two settings, the scenario was experimented in the last two settings (see table 5.4) in a 4000 step experiment which was repeated 100 times to encompass the time variations. The average values of the weights of the recall and classification methods of learning were also obtained for these two settings because on the third setting the expert and the apprentice agent face the same conditions and on the fourth setting they face different conditions. Table 5.7 presents these values.

<table>
<tr><th colspan="2">Setting</th><th>Recall Weight</th><th>Classification Weight</th></tr>
<tr><td rowspan="2">exp3</td><td>avg</td><td>0.935</td><td>0.065</td></tr>
<tr><td>stdev</td><td>0.024</td><td>0.024</td></tr>
<tr><td rowspan="2">exp4</td><td>avg</td><td>0.996</td><td>0.004</td></tr>
<tr><td>stdev</td><td>0.05</td><td>0.05</td></tr>
</table>

Table 5.7: The average weights of the recall and classification methods on each setting

Unlike what happens on the first two settings (see table 5.5), the results in table 5.7 show that when the expert and the apprentice agent face different conditions (*exp4*), the recall method has a larger influence on the proposed actions. This happens because of the problems faced by the classification algorithm and, in general, by all other supervised learning algorithms (see appendix C). When important information is missing from the training examples (in this case the visible attribute representing the state of the hand is missing from the conditions) the algorithm is not capable of proposing correct actions.

This problem caused the classification method to propose incorrect actions throughout the evaluation in the learning state (see section 4.6), which resulted in a decrease of the method's weight. This means that even though the agent is facing different conditions from the observed experts, it is more likely for the actions proposed by the recall method to be executed. On the other hand, the actions proposed by the classification method would be unreliable because of the problems faced by the missing information on the state of the hand in the conditions. Therefore, the weight mechanism plays an important role on lowering the FINALRELIABILITY of the actions proposed by the classification method, thus preventing the agent from executing those actions (see section 4.5).

Table 5.8 shows an overview of the results of experimenting the scenario in *exp3* and *exp4*. The table compares the time spent on the learning state and on the execution state, the number of actions the agent was able to execute, how many of those actions were appropriate and also the time of a simulation step when learning and when executing actions. The results in table 5.8 present both the average values and the standard deviation from running the experiment 100 times.

		Expert	Apprentice (exp3)	Apprentice (exp4)
Time spent learning (seconds)	average	-	1.97	2.39
	stdev	-	0.5	0.67
Time spent execution (seconds)	average	1.72	7.57	13.17
	stdev	0.46	1.52	0.85
Total actions executed	average	4000	3905	3696.2
	stdev	0	0	24.43
% of appropriate actions executed	average	100%	100%	11.9%
	stdev	0	0	7.9 p.p.
Step time in learning state (ms)	average	-	21.42	8.77
	stdev	-	5.62	0.87
Step time in execution state (ms)	average	0.43	2.1	3.56
	stdev	0.12	0.34	0.24

Table 5.8: Results of experimenting the last two settings of the virtual hand scenario

Table 5.8 shows that the results are similar when the apprentice agent and the expert face the same conditions both when the *hand* attribute is visible (*exp1* as shown in section 5.2.1) and when it is not visible in the software image (*exp3*). The only significant difference is the total number of actions executed, which is lower in (*exp3*). Therefore, when the expert and the apprentice agent face the same conditions, the missing information on the *hand* attribute is not enough to mislead the recall method because the agent merely follows the same sequence of

actions as it has observed in the expert.

When the expert and the apprentice agent face different conditions (*exp4*), even though the apprentice executes a similar number of actions as when the *hand* attribute is visible (*exp2* as shown in section 5.2.1), only 12% of those actions are appropriate for the faced conditions. The agent also spends less time in the learning state in *exp4* when compared with *exp2*. This makes it impossible for the agent to collect a large amount of knowledge because spending less time learning makes it harder to find snapshots that resemble the faced conditions. In addition, table 5.7 shows that the majority of the executed actions was proposed by the recall method, which requires the agent to observe a sequence of events that resemble the faced conditions.

As for the average time spent in the simulation steps, table 5.8 shows that, once again, the simulation step is longer when agents are learning because of the additional effort of discovering, observing the experts and storing the acquired knowledge (see section 5.1.1). The simulation step of the apprentice agents is also longer than the simulation step of the expert when executing actions because of the two methods for proposing actions and the agent's evaluation.

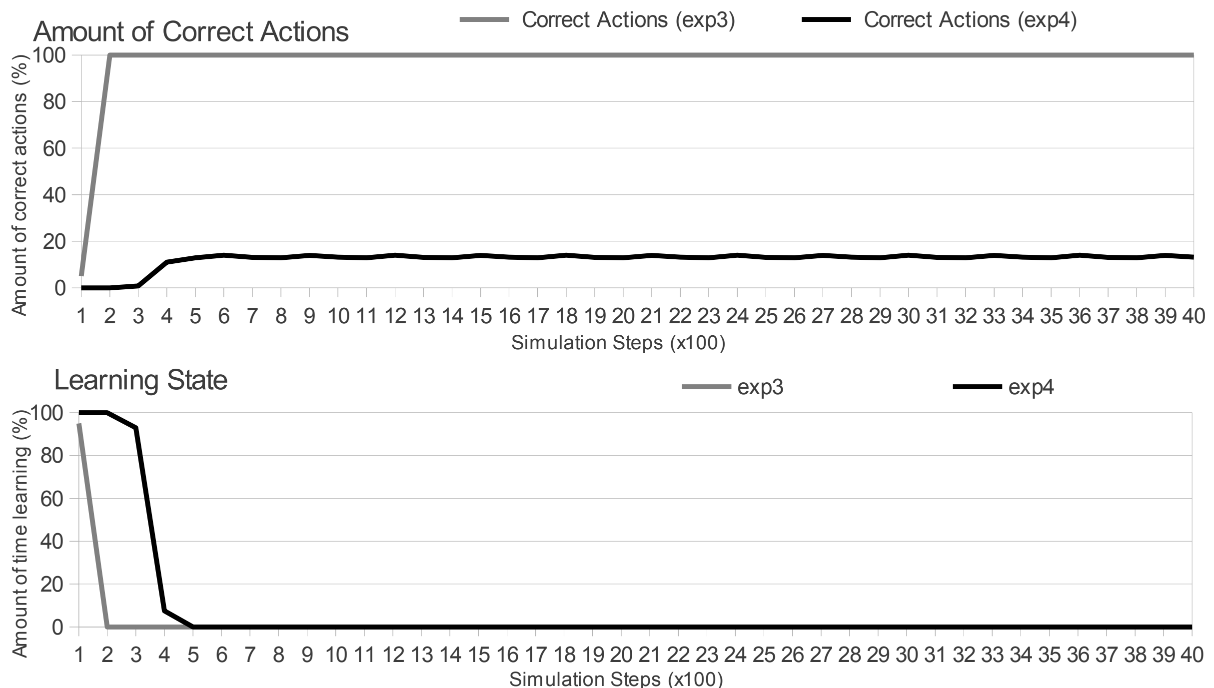


Figure 5.8: The progress of the agent in *exp3* and *exp4* throughout the experiment

The agent also spends less time in the simulation step, both when learning and when executing actions in *exp4* when compared with *exp2*. This happens because the agent acquires less information and also because the conditions in the observed data lack the information on the state of the hand (the visible attribute) and therefore the agent handles less data when storing

the acquired knowledge and when proposing actions.

A closer look at the results of the experiment for the last two settings is presented in figure 5.8, which shows the progress of the agent, in terms of the number of correct actions it has executed, throughout the experiment. The figure also shows the progress of the state of the learning process (see chapter 4) throughout the experiment. To provide a clearer presentation, the results are combined in groups of 100 simulation steps.

Figure 5.8 shows that the agent behaves in a similar way when it faces the same conditions as the expert (*exp1* and *exp3*). As for the fourth setting (*exp4*), the time spent learning is about the same time as the initial learning period in (*exp2*). The agent switches to the execution state because the recall method proposes the correct actions for the conditions faced by the expert, which increases the agent's confidence.

However, unlike what happens *exp3*, the recall method is not capable of proposing actions for the current conditions because they are different from those faced by the expert and the number of observed snapshots is not enough for the agent's memory to hold a sequence of experiences containing the current conditions. One way of increasing the experiences in the agent's memory is through subsequent learning periods, which are not happening in this setting.

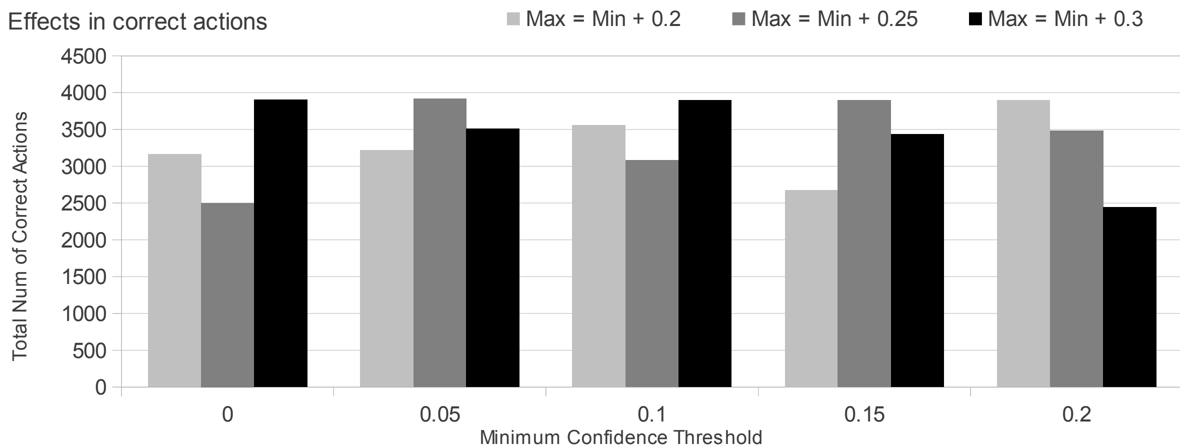
The agent also has no subsequent learning periods in *exp4* because the conditions in the observed snapshots only contain the number provided by the source and therefore the amount of different combinations of conditions is smaller. With a smaller set of different conditions it becomes harder for the agent's evaluation to detect when it is facing unfamiliar conditions and thus force the agent to return to the learning state (see section 4.6). The agent's evaluation also has no way of determining if the agent is executing the appropriate actions for the faced conditions and thus influence the agent's confidence.

One way of overcoming this problem is using external feedback (see section 4.6). Section 5.2.4 shows how the external feedback increases the agent's performance in terms of the amount of appropriate actions executed in this case.

5.2.3 Setting the Configurable Parameters

One of the experiments with the virtual hand scenario is determining the impact of changing the values for the UPPERCONFIDENCETHRESHOLD and the LOWERCONFIDENCETHRESHOLD (see section 4.6). For this reason, the first and second settings of the scenario (see table 5.4) were tested on agents using different values for these thresholds. Figure 5.9 presents a summary

of the results obtained in the second setting in experiments with 4000 steps (each variation of the thresholds is tested in an experiment that lasts 4000 steps). The reason for selecting the results for the second setting is because it is where changing the thresholds has more influence and also because it is a more realistic approach for the scenario, since the chances for the apprentice agent and the expert to face the same conditions are very small.



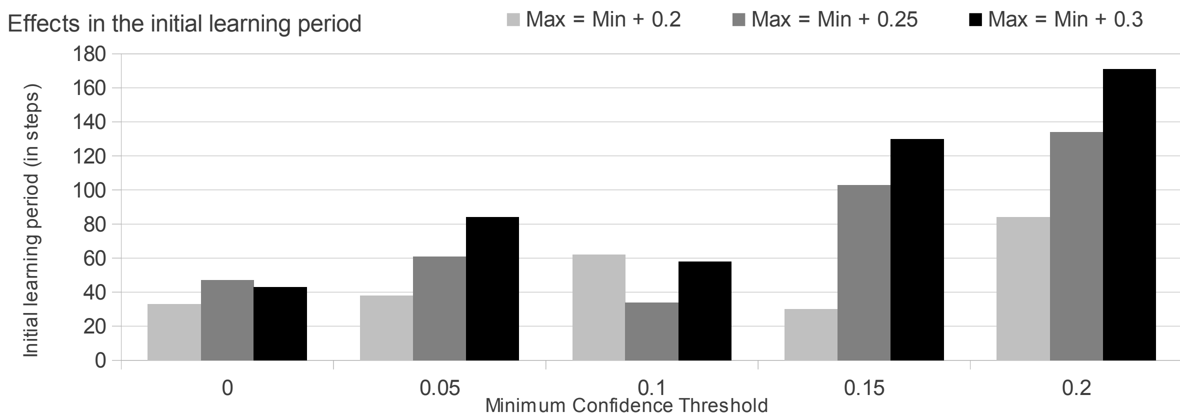


Figure 5.9: The impact of changing the confidence thresholds

As explained in section 4.6, the confidence thresholds affect the length of the learning period and the number of subsequent learning periods which in turn affects the total number of correct actions performed by the agent throughout the experiment. When the UPPERCONFIDENCETHRESHOLD is low, the agent may not have enough time to learn, which decreases the agent's performance in terms of being able to execute the correct actions. When the UPPERCONFIDENCETHRESHOLD is too high the agent spends a lot of time learning, which reduces the time spent in the execution state and therefore the total number of actions executed by the agent is lower.

When the LOWERCONFIDENCETHRESHOLD is too close to the UPPERCONFIDENCETHRESHOLD the agent switches between the learning and execution state more often (see chapter 4). This may hinder the agent‘s ability to complete a task because the slightest error causes the agent to return to the learning state. When the LOWERCONFIDENCETHRESHOLD is too far apart from the UPPERCONFIDENCETHRESHOLD, the agent takes longer to switch between learning and execution. This slows down the ability to recognize the mistakes and switching to the learning state. It also slows down the recovery from the learning state to the execution state.

The results presented in figure 5.9 fall in the region where the LOWERCONFIDENCETHRESHOLD changes between 0 and 0.2 and the UPPERCONFIDENCETHRESHOLD changes between 0.2 and 0.3 units more than the LOWERCONFIDENCETHRESHOLD. This is where the agent is able to perform the maximum number of correct actions in the time spent by the experiment (4000 steps). If the difference between the thresholds is larger than 0.3, the total number of correct actions decreases since the agent takes longer to change between the learning and execution states.

Figure 5.9 also shows that the length of the initial learning period increases as the LOWERCONFIDENCETHRESHOLD increases. The longer the agent spends learning the less time it has to perform the actions. Therefore, as Figure 5.9 shows, the optimal values (for this scenario) for the thresholds are 0.05 for the LOWERCONFIDENCETHRESHOLD and 0.3 for the UPPERCONFIDENCETHRESHOLD. This provides the agent with a learning period that is long enough for the agent to learn all the necessary skills (so that the majority of the actions it performs are correct) but short enough to allow the agent to perform the maximum amount of actions (which is 3920 actions according to figure 5.9).

In addition to the LOWERCONFIDENCETHRESHOLD and the UPPERCONFIDENCETHRESHOLD, the AMOUNTUNFAMILIARCONDITIONS and the UNFAMILARCONFIDENCEREFERENCE are also configurable parameters of the proposed approach (see section 4.6). Additional tests on these configurable parameters show that the AMOUNTUNFAMILIARCONDITIONS must have a low value (below 10) and the UNFAMILARCONFIDENCEREFERENCE must not be much lower than the LOWERCONFIDENCETHRESHOLD, usually not more than 0.03 units below.

A low value for the AMOUNTUNFAMILIARCONDITIONS allows the agent to go back learning after being faced with a small and uninterrupted sequence of unfamiliar conditions. When the UNFAMILARCONFIDENCEREFERENCE is only a little lower than the lower confidence threshold, the agent spends less time to regain confidence on its knowledge and to go back

executing actions. The tests show that, for this scenario, the appropriate value for the AMOUNTUNFAMILIARCONDITIONS is three and the appropriate value for the UNFAMILARCONFIDENCEREFERENCE is 0.04 (0.01 units bellow the reference LOWERCONFIDENCETHRESHOLD).

Additional experiments also revealed that the size of the history record (see section 3.2.2) and the timeout for the observation period (see section 4.3) have little influence in the agent's performance in the tested scenarios. Therefore, all scenarios were experimented with the reference values for these configurable thresholds: ten snapshots for the size of the history record and fifty snapshots for the observation period.

In addition to determining the impact of the configurable threshold, the virtual hand scenario was also the background for experimenting the influence of the external feedback (see section 5.2.4) and the capabilities of the software image meta-ontology when comparing software agents (see section 5.2.5).

5.2.4 The Influence of the External Feedback

This section shows how the external feedback provided by a specialized expert influences the agent's ability to execute appropriate actions in the fourth setting of the scenario (*exp4* in table 5.4). In this new setting, *exp5*, the expert and the apprentice agent face different conditions, the *hand* attribute is not visible and one of the experts is a teacher. The scenario was experimented in a 4000 step experiment which was repeated 100 times to encompass the time variations. Table 5.9 shows an overview of the results for this setting.

	Time spent learning (s)	Time spent execution (s)	Total actions executed	Amount of approp. actions
avg	19	22.37	3515.9	81.8%
std	14.7	7.4	200.3	11 p.p.
	Step time in learning state (ms)	**Step time in execution state (ms)**	**Recall weight**	**Classification weight**
avg	14.47	6.35	99%	1%
std	10.9	2.01	4.9 p.p.	4.9 p.p.

Table 5.9: Results of experimenting the fifth setting of the virtual hand scenario

Table 5.9 shows that, when compared with the fourth setting (*exp4*), the teacher increases the agent's performance, in terms of the amount of actions correctly executed (up to 82% of the executed actions are correct), without affecting the total amount of executed actions. The

teacher provides the agent's evaluation with additional feedback on the executed actions, indicating those that were appropriate or inappropriate for the faced conditions. The feedback influences the agent's confidence which allows the agent to return to the learning state more often, and acquire a broader set of experiences from the experts.

Table 5.9 also shows that, as in *exp4*, the majority of the executed actions is proposed by the recall method. In the case of *exp5* the teacher feedback affects the time the agent spends learning, which is longer than in *exp4*. Therefore, the agent acquires more knowledge which makes it more likely to find a sequence of experiences, in the agent's memory, whose conditions are similar to faced the conditions. The high standard deviation values for both the time spent learning and the simulation step time in the learning state is the result of the randomness of the observed experts. As in the previous settings, the agent observes different subsets of the available experts in a different sequence, which changes the knowledge retained by the agent.

Given that the agent acquires more knowledge, the time taken by a simulation step is larger than in *exp4* because the agent handles more data when proposing the actions. Table 5.9 shows that, in comparison with the other settings (*exp1*,*exp2* and *exp3*) using a teacher does not increase the simulation step time when the agent is in the execution state. The amount of data acquired when observing has a much larger influence in the simulation step time in the execution state, as the results for *exp2* in table 5.6 show.

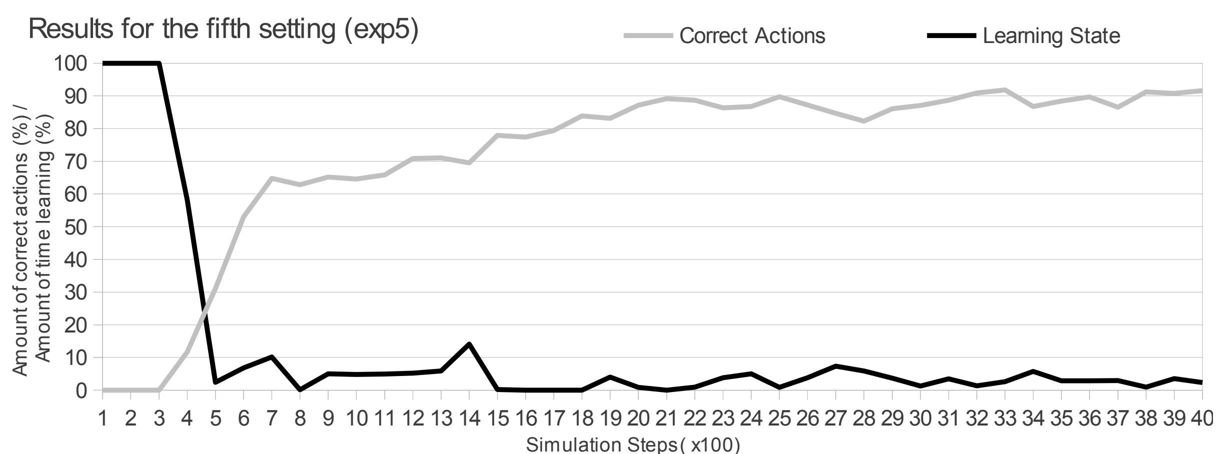


Figure 5.10: The progress of the agent in *exp5* throughout the experiment

A closer look at the results of the experiment for *exp5* is presented in figure 5.10, which shows the progress of the agent, in terms of the number of correct actions it has executed, throughout the experiment. The figure also shows the progress of the state of the learning process (see chapter 4) throughout the experiment. To provide a clearer presentation, the results

are combined in groups of 100 simulation steps.

As figure 5.10 shows, unlike in *exp4*, the amount of correct actions has the tendency to grow as the experiment advances. The teacher allowed the agent to return to the learning state for several times throughout the experiment, which provided the recall method with a greater amount of sequences of experiences (see section 4.5.1). Even though the agent returned to the learning state several times throughout the experiment, these subsequent learning periods represent less than 10% of the simulation steps.

In addition, the teacher also prevented the apprentice from executing incorrect actions that could lead to situations that would never be observed in the expert and from where the apprentice would never know how to get out. For example, a hand with the pinky finger up and all other fingers down does not represent a number so the expert hand structure would never present that state. The results from this setting show that using external feedback in the agent's evaluation increase the agent's performance in terms of the actions executed correctly with minimal effects on the total number of actions executed by the agent and on the time spent executing those actions.

5.2.5 The Impact of the Software Image Meta-ontology

This section shows two experiments with the virtual hand scenario that show the capabilities of the software image meta-ontology, especially the abilities concerning the comparison of software images (see section 3.4.3). The first experiment shows the way the agent discovers an expert whose static image contains the elements that are necessary to accomplish a task. The second experiment shows the way the agent observes an expert whose *hand* visible attribute has a different designation.

In the first experiment, the agent has to discover and observe an expert that only shares part of the constituents and capabilities of the agent. Figure 5.11 shows a representation of the agent parts that are shared by both agents and of the ontology that describes the knowledge on the designations of those parts.

The expert (*expert1*) has a specialized part which holds the knowledge on the most efficient way of changing the virtual hand. The part has the same constituents as the agent, that is, a sensor that perceives numbers from the environment, a *hand* visible attribute that shows the current state of the agent's virtual hand and one actuator with five actions that change the state of each finger of the virtual hand. Another specialized part holds the knowledge on how to

interact with a graphical interface, whose constituents are different from those of the agent.

Figure 5.11 shows that the ontology indicates that all the elements that are shared by the agent and *expert1* are necessary to perform the task *handleHand*. As explained in section 4.3.1, the agent can observe an expert as long as it has the elements that are necessary for the task to accomplish. Therefore, it is possible for the agent to observe this expert and learn how to perform the task of handling the hand.

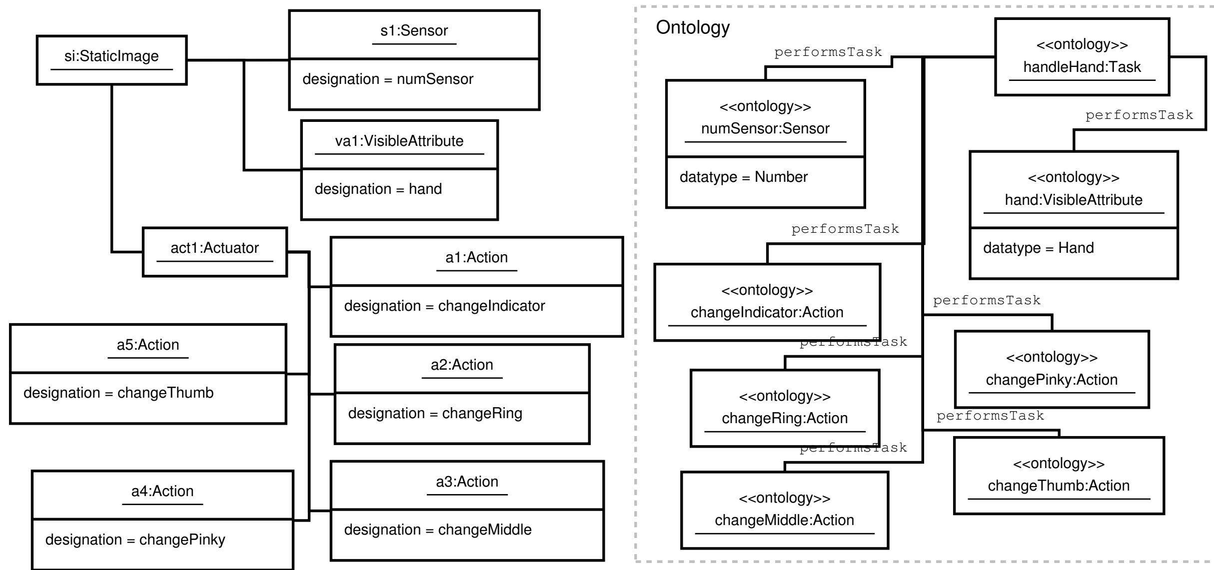


Figure 5.11: The parts shared by the agent and *expert1*

Table 5.10 shows a comparison of learning from *expert1* and from an identical expert that has the same constituents and capabilities (*expert2*). The experiment for this comparison lasts 200 steps and both the experts and the apprentice agent face the same conditions. The experiments were repeated 100 times to encompass the time variations that might exist in the experiments. Table 5.10 shows the average values and the standard deviation of the time spent learning, the total actions executed and the amount of correct actions executed by the agent.

		expert1	*expert2*
Time spent learning (seconds)	average	2.92	1.98
	stdev	0.55	0.51
Total actions executed	average	104	104
	stdev	0	0

Table 5.10: Comparison between learning from *expert1* and from *expert2*

The results from table 5.10 show that not only it is possible to learn from an expert with only one identical part but also there are no significant differences between learning from an identical expert and from an expert containing the elements that are necessary to perform a task.

In both cases, the agent was able to perform the same amount of actions and all of the executed actions were appropriate for the faced conditions.

The only difference between the two cases is in the time spent learning, which is 50% larger when learning from an expert containing the elements that are necessary for the task to learn. This was expected because of the extra processing required when comparing and observing such experts. As explained in section 4.3, when agents observe such kind of experts they have to initially identify the elements required to perform a task and determine if the expert contains those elements. When observing, the agent also has to remove all unnecessary conditions.

In addition to experimenting learning from an expert containing the elements necessary for a task, the virtual hand scenario was also used to experiment ontology translation using the *equivalentTo* relationship of the software image meta-ontology (see section 3.3). In this experiment, the agent has to discover and observe an expert with a different designation and representation for the *hand* visible attribute. Figure 5.12 shows the differences between the two designations for the visible attribute.

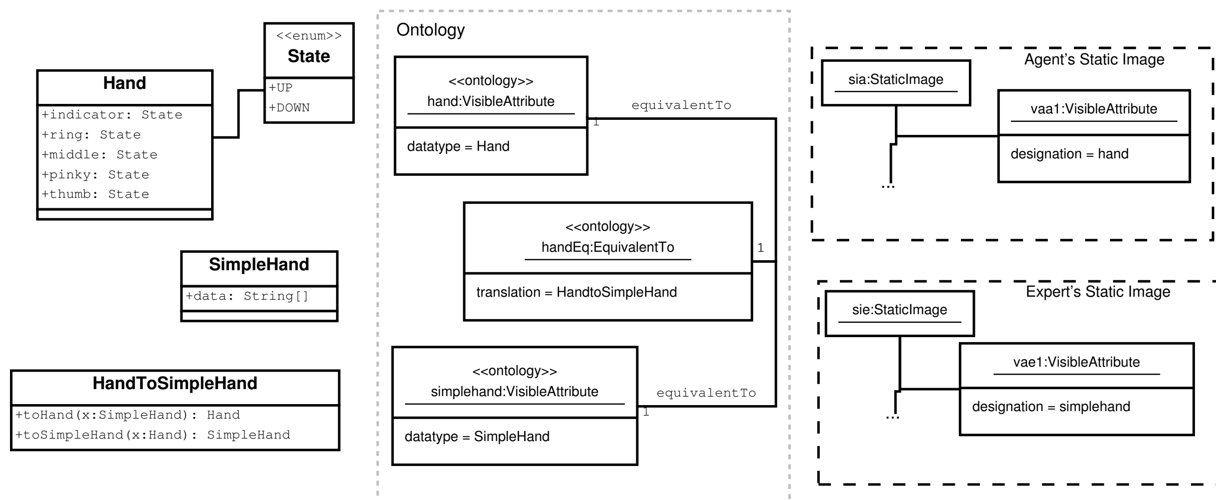


Figure 5.12: Representation of the different designations for the visible attribute

As figure 5.12 shows, the ontology states that the *simpleHand* visible attribute of the expert is equivalent to the *hand* visible attribute of the agent, even though they represent the state of the virtual hand with different kinds of data. The *equivalentTo* relationship defines a translation class, the *HandToSimpleHand*, which converts the instances of the visible attribute from one kind to the other. This translation class is essential for the agent to understand the information provided by the expert's visible attribute (see section 3.3).

Table 5.11 shows a comparison between the time it takes to discover and observe this expert and an expert with an identical visible attribute. The comparison was repeated 100 times to

encompass the time variations. The table shows the average values and the standard deviation of the time spent discovering the agent and the time spent observing one snapshot on both experts.

		Identical Visible Att.	Equivalent Visible Att.
Time spent discovering (s)	average	1.47	1.48
	stdev	0.2	0.2
Time spent observing (ms)	average	15.54	15.6
	stdev	2.3	2.4

Table 5.11: Comparison between learning from a task and from an identical agent

The results in table 5.11 show that there are no differences between discovering and observing an identical expert and an expert with an equivalent visible attribute that needs to be translated. Even though discovering and observing an identical expert is, on average, a little faster, the difference falls in the standard deviation which makes it statistically irrelevant. As for the ability to learn from the expert with an equivalent visible attribute, the results obtained from experimenting all the settings with those experts are similar to the results obtained from experimenting the settings with identical experts.

The following sections compare the proposed approach with other approaches to learning.

5.3 The Calculator Scenario

This section describes the second scenario in which the proposed approach was implemented. The scenario compares the proposed learning approach with two supervised learning algorithms (KStar and multi-layer perceptron back-propagation) that are limited to observing the visible effects of the agent actions on the state of the environment in a situation where only some of these effects are visible.

In this scenario an agent learns how to calculate mathematical expressions composed of numbers and operations acquired from the environment. The software environment is composed of generators that provide the numbers (from 0 to 9) and the operations to be performed (addition, subtraction, multiplication and division). The numbers and the operators are acquired from the environment, one at a time, and combined to create a mathematical expression that is calculated by the agents. For example, when an agent acquires the sequence $(1, +, 1, =)$ it has to calculate the expression $1 + 1$ and output its result.

The experts, provided for this scenario, know the optimal way of acquiring, retaining and calculating the acquired expressions. The experts acquire the numbers and operations using specialized sensors. Depending on what is acquired, the experts can store the acquired number, perform a calculation or output a result. For example, if the expert acquires the number 1 and then the number 5, it stores 1 in the memory and then joins 5, which results in having 15 in memory. If the expert perceives 1 followed by the plus sign (addition) followed by 5 it stores 1 in the memory and then adds 5 to the memory, which results in having 6 in memory. When experts acquire the equals sign they output the number stored in their memories.

The effect of acquiring the numbers and operations is only partially visible in the environment. The visible part of the effect of these actions is the disappearance of the number or the operator from the environment, which is the same for all actions. The invisible part of the effect is the processing in the agent's memory. The agents perform several calculations when acquiring the numbers and operators and store the results in their memories. The only action whose effects are completely visible in the environment is the output of the result stored in the agent's memory, which only happens when the equals sign is acquired. Therefore, observing only the visible effects of the actions in the environment makes it harder for agents to learn.

Apprentice agents have to learn how to build and calculate the expressions from the acquired numbers and operators to provide the correct outputs. The scenario includes three kinds of apprentice agents to compare the proposed approach with two implementations of supervised learning algorithms. The first kind of apprentice uses the proposed approach to learning by observation. The second kind of apprentice uses a KStar supervised learning algorithm implemented in the WEKA software workbench (Hall *et al.*, 2009). The third kind of apprentice uses a multi-layer perceptron back-propagation classifier (MLP) also implemented in the WEKA software workbench (Hall *et al.*, 2009).

The apprentice agents have the same constituents and capabilities of the experts, a single part with two sensors and two actuators. One of the sensors perceives the numbers from the environment and the other perceives the operations. The first actuator provides access to the agent's memory and outputs the stored number. The second actuator acquires the numbers and operators from the environment and performs five operations on numbers, the four arithmetic operations (addition, subtraction, multiplication and division) and joining two numbers (for example, the result of joining 1 with 1 is 11). Additional information on the scenario setting is presented in appendix D.

The configurable parameters of the learning by observation agent were set with the optimal values for this experiment: 0.4 for the UPPERCONFIDENCETHRESHOLD, 0.05 for the LOWERCONFIDENCETHRESHOLD, 3 for the AMOUNTUNFAMILIARCONDITIONS and 0.04 for the UNFAMILARCONFIDENCEREFERENCE. Section 5.2.3 explains the way these optimal values are obtained.

To reduce the accumulation of the effects of the actions in the environment, the agents were tested individually with a single expert performing the task. Both the learning by observation and the supervised learning agents acquire their training data from the same kind of expert performing the same task. The learning by observation agent learns by observing the actions executed by the expert while performing that task. The supervised learning agents learn (acquire their training data) by observing the changes on the state of the environment while the expert performs the same task.

As with all supervised learning approaches, the two supervised learning agents have an initial learning period where they acquire all the necessary training data. This initial learning period lasts the same number of simulation steps as the learning by observation agent takes in the learning state. This ensures that both the learning by observation and the supervised learning agents acquire a similar amount of training examples, even though one of them observes the actions while the others observe their effects.

The scenario was experimented in two different settings. In the first setting both the expert and the apprentice agent face the same conditions, that is, they acquire the same numbers and operations in the same sequence throughout the experiment. In the second setting, the expert and the apprentice agent face different conditions, that is, the apprentice acquires a different sequence of numbers and operators throughout the experiment. Both settings last 4000 simulation steps and were repeated 100 times to encompass the time variations.

The results of the experiments present a comparison of the time the agents spent learning and executing actions, the amount of expressions they were able to calculate and how many of those calculations were correct. Table 5.12 presents a summary of these results for the first setting, that is, when the expert and the apprentice agents face the same conditions. The student's t-test was used to ensure the statistical relevance of the comparison.

Table 5.12 shows that, as expected, the expert agent exhibits the best results in all aspects. The table also shows that the learning by observation agent was the only one who learned how to perform the task correctly, outperforming the two supervised learning agents (KStar and

multi-layer perceptron). Even though the KStar algorithm spends less time executing actions, more than 50% of the task is incorrect. The learning by observation agent spends more time executing actions than the KStar agent because it uses two methods to propose actions whereas the KStar algorithm only uses one.

	Time spent		Expressions	% correct
	learning (s)	executing (s)	calculated	outputs
Expert	-	1.13	2824	100%
Agent (LbO)	8.37	5.07	2794	100%
Agent (KStar)	0.12	3.27	2794	48.97%
T-TEST (LbO-Kstar)	4×10^{-7}	2.8×10^{-6}	0	0
Agent (MLP)	0.14	40745	2794	48.97%
T-TEST (LbO-MLP)	4×10^{-9}	1×10^{-49}	0	0

Table 5.12: Overview of the results of the experiment of the number calculator scenario when facing the same conditions

The multi-layer perceptron agent takes more time to execute the actions because the knowledge is organized in a way that increases the algorithm's recovery capacity, which requires a larger amount of processing when executing actions (Hall *et al.*, 2009). Even with an increased recovery capacity, more than 50% of the task performed by the multi-layer perceptron agent is incorrect.

Table 5.12 also shows that both the KStar and the multi-layer perceptron agents spend less time learning than the learning by observation agent even though they acquire the same number of training examples. The learning by observation agent spends more time learning because it performs additional tasks such as discovering the experts to observe and evaluating its knowledge. Even though it spends more time, the learning by observation agent is the only one who can perform all the actions correctly.

Even though the KStar and the multi-layer perceptron agents calculate the same amount of expressions as the learning by observation agent (2794), more than 50% of their outputs are incorrect. This proves that even though facing the same conditions as the expert, the tested supervised learning algorithms cannot provide reliable results.

Figure 5.13 shows the progress of the apprentice agents (learning by observation, KStar and multi-layer perceptron) throughout the experiment in terms of the amount of correct outputs and the time spent learning. For a clearer presentation, the results are combined in groups of 100 simulation steps.

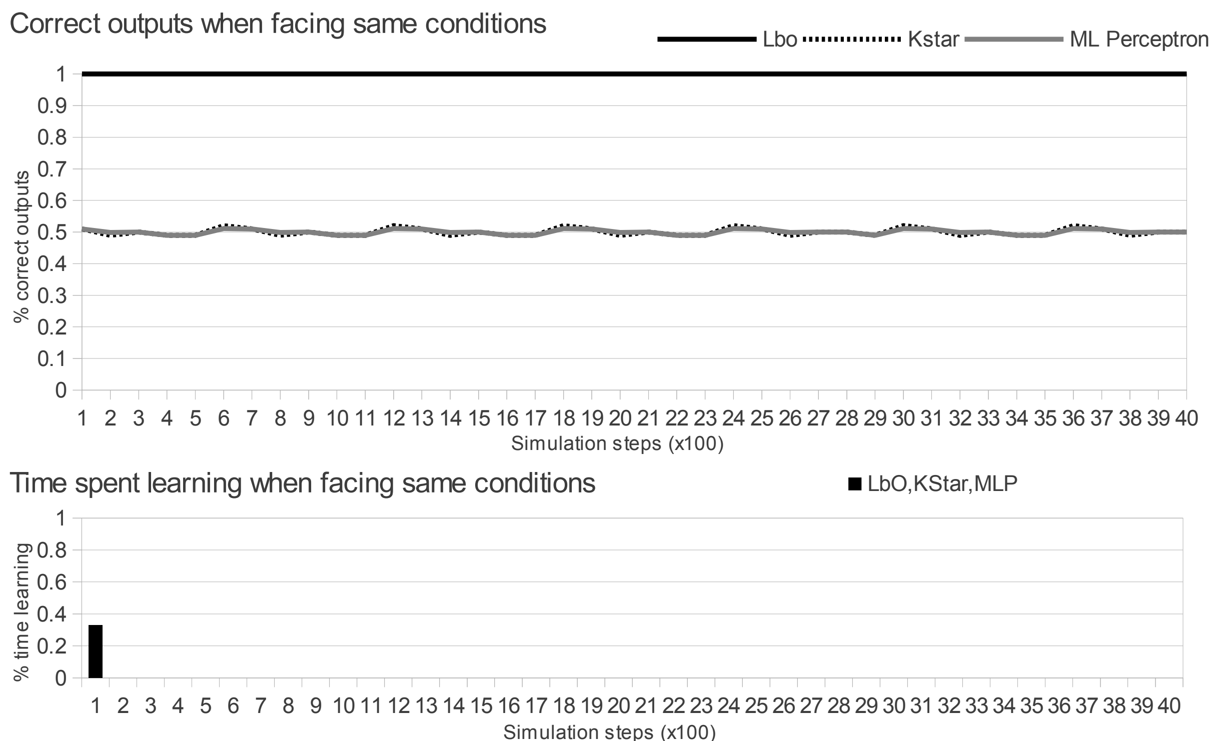


Figure 5.13: Progress of the apprentice agents for the first setting of the calculator scenario

Figure 5.13 shows that the learning by observation agent provides correct outputs throughout all the experiment whereas only between 49% and 51% of the outputs provided by the KStar and the multi-layer perceptron agents are correct. The figure also shows that all agents have the same behaviour regarding learning. The agents require only 33% of the first 100 steps to learn. For the rest of the experiment the agents apply the knowledge they acquired by executing actions. The initial steps spent learning also explain the difference between the amount of expressions calculated by the expert and those calculated by the apprentice agents because while the agents were learning the expert was already performing calculations.

	Time spent		Expressions	% correct
	learning (s)	executing (s)	calculated	outputs
Expert	-	1.13	2824	100%
Agent (LbO)	391.9	9.115	2710	98.9%
Agent (KStar)	0.13	3.07	2710	4.07%
T-TEST (LbO-Kstar)	1×10^{-31}	3.2×10^{-18}	0	0
Agent (MLP)	0.12	34091	2710	12.2%
T-TEST (LbO-MLP)	1.4×10^{-31}	1.3×10^{-56}	0	0

Table 5.13: Overview of the results of experimenting the number calculator scenario when facing different conditions

Table 5.13 presents a summary of the results for the second setting, where the expert and the apprentice agents face different conditions. The second setting shows an example where the tested supervised learning algorithms are commonly used, which is when the agent faces a problem that is different from the problem presented in the training examples. The table presents a comparison based on the time the agents spent learning and executing actions, the amount of expressions they were able to calculate and how many outputs of those calculations were correct. The data in table 5.13 was obtained from running the scenario 100 times to encompass time variations. The student's t-test was used to ensure the statistical relevance of the comparison.

Table 5.13 shows that the learning by observation agent is, once again, the one who provides the highest number of correct outputs (about 99% of them are correct). When compared with facing the same conditions (see table 5.12), the amount of correct outputs of both the KStar and the multi-layer perceptron agents decreases when the agents face different conditions. The same does not apply to the learning by observation agent, whose effect on the number of correct outputs is much smaller (less than 1%). This shows that facing different conditions has a larger impact on the tested supervised learning algorithms than on the learning by observation agent.

Table 5.13 also shows that the multi-layer perceptron agent has a better recovery capacity than the KStar agent, even though only 12% of its outputs are correct. Once again, the KStar algorithm takes less time to execute the actions but only about 4% of its outputs are correct. The learning by observation agent takes longer than the KStar algorithm to execute but provides a larger number of correct outputs. The multi-layer perceptron agent takes even longer to execute the actions because of the way the knowledge is organized (Hall *et al.*, 2009).

For the learning by observation agent, facing different conditions only affects the time spent learning and the time spent executing actions (see table 5.13) when compared with facing the same conditions as the expert (see table 5.12). The agent spends more time learning and also executing actions, even though it calculates fewer expressions than when facing the same conditions (2710 expressions instead of the 2794 expressions, as shown in table 5.12). The increased amount of training examples is the reason for the increase in the time spent learning and executing actions because it requires more processing (see section 4.5).

Figure 5.14 shows the progress of the apprentice agents (learning by observation, KStar and multi-layer perceptron) throughout the experiment in terms of the number of correct outputs and the number of simulation steps spent learning. For a clearer presentation, the results are

combined in groups of 100 simulation steps.

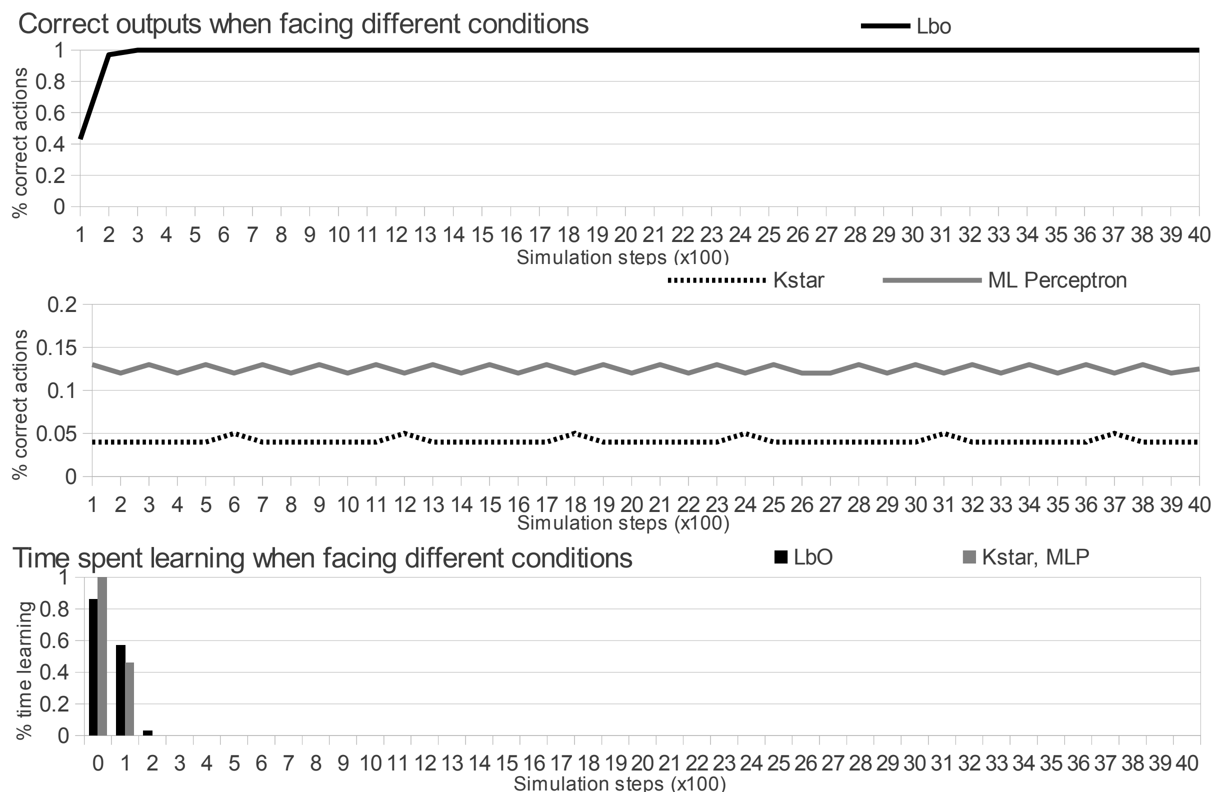


Figure 5.14: Progress of the apprentice agents for the second setting of the calculator scenario

Figure 5.14 shows that, when compared with the first setting, the agents spend more simulation steps learning. Spending more time learning allows the agents to acquire more examples which increases the diversity of the agent's knowledge. The effect of spending more time learning is larger in the learning by observation agent because of the amount of additional processing, which includes using two methods to propose actions, evaluate the proposed actions and store the knowledge (see chapter 4), which does not happen for the two supervised learning agents.

Figure 5.14 also shows that all agents spend the same amount of simulation steps when learning (about 146 steps). The learning by observation agent has an initial learning period of about 86 steps which is followed by several subsequent short term learning periods. The incorrect outputs provided by the learning by observation agent come from this initial period, which explains the decrease of the amount of correct outputs in table 5.13. Nevertheless, after that initial period the outputs provided by the agent are all correct as what happens when the apprentice and the expert agent face the same conditions.

An important aspect to take into account when comparing the two settings is the average time spent in a simulation step. Table 5.14 shows the average times and the standard devi-

ation for all agents in both settings. The results in table 5.14 were obtained by running the experiments 100 times to encompass time variations.

Simulation step times (ms)		Facing same conditions		Facing different conditions	
		Learning	Executing	Learning	Executing
Expert	avg	-	0.28	-	0.28
	stdev	-	0.08	-	0.08
Agent (LbO)	avg	179.7	1.2	233.3	2.31
	stdev	19.3	0.13	9.7	0.19
Agent (KStar)	avg	1.45	0.83	1.45	0.78
	stdev	0.41	0.1	0.41	0.23
Agent (MLP)	avg	1.68	10394	1.35	8698
	stdev	0.6	31.9	0.3	12.9

Table 5.14: The time spent by a simulation step for the calculator scenario

Table 5.14 shows that the KStar agent has the shortest simulation steps. The simulation steps of the multi-layer perceptron agent are similar to those of the KStar agent when learning, but when executing, the simulation steps are the longest because of the way the knowledge is organized. The learning by observation agent has the second longest simulation step when executing because it uses two methods for proposing actions (see section 4.5) and evaluates the executed actions (see section 4.6) whereas the KStar agent only uses one method and does not evaluate the executed actions.

The learning by observation agent also has the longest simulation step when learning because of the additional processing of the proposed approach. This processing does not exist in the tested supervised learning agents (KStar and multi-layer perceptron) and part of it, especially the agent's evaluation (see section 4.6), is what allows the agent to overcome the others in terms of the amount of correct outputs (see tables 5.12 and 5.13).

Figure 5.14 shows that when agents face different conditions they spend more simulation steps learning and therefore can acquire more examples. However, this does not mean that all agents take benefit of that and are able to acquire a larger diversity of knowledge from those examples.

Being able to only observe the effects of actions may reduce the variety of the examples to the point that increasing the amount of examples acquired in learning has no effect on the amount of knowledge acquired because many examples will be identical to those already acquired. The small difference in the simulation steps when facing the same conditions and when facing different conditions from the expert is a good indicator that, for both the KStar and the multi-layer perceptron agents, increasing the amount of examples does not increase the acquired

knowledge.

In addition to comparing the proposed approach with common supervised learning algorithms, the application scenarios also compare it with other learning techniques such as reinforcement learning, as described in the following section.

5.4 The Mountain Car Scenario

This section describes the third scenario in which the proposed approach was tested. The scenario was proposed by Sutton and Barto (Sutton & Barto, 1998) and is a software implementation of an agent that learns how to climb a mountain simulated by a sinusoidal wave. The agent must abide by the laws of physics to climb the mountain, and because it has no sufficient force, it will not be able to climb the mountain by going forward only. It needs to accelerate backwards and forwards to gain momentum. The goal of this scenario is to reach the top of the mountain, that is, the peak of the sinusoidal wave, taking the least number of decisions and travelling the smallest distance (up and down the mountain) as possible.

The experts provided for this scenario know the optimal way (the exact moment and direction they need to accelerate) to climb the mountain and reach its top. The experts perceive their current speed and direction and their location in the mountain through their sensors and use this information to decide the direction with which they should accelerate next. They can choose between accelerating forward, accelerating backward or not accelerating (which maintains their current speed).

The apprentice agents have to learn to decide which direction to accelerate according to their location, speed and direction. The scenario includes two kinds of apprentice agents to compare two different learning methods. The first kind of apprentice uses a reinforcement learning algorithm (Q-Learning, implemented in the PIQLE tool (Comité, 2005)). The reinforcement learning agent is able to perform three actions, accelerate forward, accelerate backwards or maintain speed.

The reinforcement learning agent is configured with a learning rate $\alpha = 0.2$ and a discount factor/rate $\gamma = 0.9$, which are the settings that present the best results. The learning rate also decreases with time following a geometrical decay, which allows the agent to eventually stop learning after a period of time. A simple reward scheme is used for the reinforcement learning algorithm. The agent is only rewarded when it reaches the goal of this scenario, the top of the

mountain.

Any other reward schemes would require the development of some kind of supervisor to determine the ideal situations to apply the reinforcements. This would have changed the comparison we intended for this scenario, which is to compare learning by observation with a situation where there is no effort for developing supervisors or reward schemes. Even though this specific scenario required the development of expert agents, in a typical application of learning by observation, the experts already exist. The only situation in reinforcement learning where there is no effort in developing reward schemes is with this simple reward. The only information provided is the goal, which is embedded in the agent.

The second kind of apprentice agent uses learning by observation to acquire the knowledge about climbing the mountain. The agent has the same constituents and capabilities of the experts it observes: a single part with two sensors and one actuator. The sensors provide the agent with its location in the mountain and its current speed (which can be positive if the agent is moving forwards or negative if the agent is moving backwards). The actuator provides the agent with three actions, accelerate forwards, accelerate backwards and maintain speed. Additional information on the scenario setting is presented in appendix D.

The configurable parameters of the learning by observation agent were set with the optimal values for this experiment: 0.45 for the UPPERCONFIDENCETHRESHOLD, 0.1 for the LOWERCONFIDENCETHRESHOLD, 3 for the AMOUNTUNFAMILIARCONDITIONS and 0.08 for the UNFAMILARCONFIDENCEREFERENCE. Section 5.2.3 explains the way these optimal values are obtained.

The scenario was experimented in a setting where all the participant agents face the same conditions. All agents are placed in identical mountains at the same starting point, about 1 meter away from the top of the mountain. At this point, moving forwards only does not provide enough momentum for the agent to reach the top. The agent has to move backwards first. The results of the experiment present a comparison of the number of actions, the distance travelled and the time spent until reaching the top of the mountain (the goal of the experiment). The results also determine the average time of a simulation step for each agent and how much time it takes for an agent to learn the task, that is, how much time it takes for an agent to reach the top of the mountain for the first time.

Since this scenario provides a goal, the time-frame of the experiment is grouped in attempts of achieving the goal. Each attempt lasts a variable number of simulation steps, with a maximum

duration of 500 simulation steps. If the agent achieves the goal before the 500 steps the attempt is completed and is regarded as successful. If after the 500 steps the goal is not achieved, the attempt is regarded as failed. The experiment lasts 50000 attempts, to provide the reinforcement learning agents with enough time for testing all the hypotheses and provide their best results.

Table 5.15 presents a summary of the important aspects of the scenario such as the amount of time, number of attempts, number of actions and distance travelled by each agent to reach the top for the first time. The table also shows how many times the agent reached the top of the mountain, the average time of a simulation step and the average number of simulation steps in an attempt to reach the goal. The data in table 5.15 was obtained from running the scenario 100 times to encompass time variations. The student's t-test was used to ensure the statistical relevance of the comparison of the two agents.

	Expert	Agent (LbO)	Agent (RL)	T-TEST (LbO - RL)
Time to reach top first time (ns)	30	3300	14660	2.2×10^{-7}
Attempts for reaching top first time	1	1	105	3.4×10^{-10}
Actions executed to reach top first time	136	278	52471	3.2×10^{-10}
Distance travelled to reach top first time (meters)	2.73	2.73	610.37	2.5×10^{-9}
Number of times reached top	50000	50000	49895	2.9×10^{-10}
Average simulation step time (ms)	0.28	4.94	0.31	2.7×10^{-39}
Average simulation steps spent in an attempt	136	136.01	229.92	4.9×10^{-12}

Table 5.15: Overview of the results of experimenting the mountain car scenario

Table 5.15 shows that, as expected, the expert agent exhibits the best results in all aspects. The table also shows that the learning by observation agent outperforms the reinforcement learning agent in all aspects with the exception of the time of a simulation step. The learning by observation agent also gets close to the same results as the expert in all aspects with the exception of the time it takes to reach the top and of the simulation step time.

When compared with the reinforcement learning agent, the simulation step of a learning by observation agent lasts longer because of the amount of processing behind that decision. The

learning by observation agent uses two methods for proposing the actions (the recall and classification methods as explained in section 4.5) which requires an additional effort of combining the results and choosing the best action from them. The agent also has to perform a set of other tasks, like for example the agent's evaluation (see section 4.6). Nevertheless, the learning by observation agent learns faster and is able to reach the goal first. It also requires less simulation steps to reach the goal after learning how to do it.

Although the simulation step of a reinforcement learning agent takes less time, the agent requires more steps and attempts to reach the top of the mountain for the first time. It also takes more time to learn how to reach the top of the mountain as the results in table 5.15 show. This puts the reinforcement learning agent in last place when considering the time it takes to reach the top for the first time. The results of the t-test in table 5.15 show that the data acquired in the experiments is statistically relevant.

In addition to the information on achieving the goal for the first time, it is also important to know how the agents performed throughout the experiment. Figure 5.15 shows the progress of the expert and the two apprentice agents (learning by observation and reinforcement learning) throughout the experiment in terms of the number of simulation steps required to reach the top. For a clearer presentation, the results are combined in groups of 100 attempts to reach the goal. Each attempt lasts a number of simulation steps that ranges from 136 to 500, depending on the number of steps that are necessary to reach the top of the mountain (see table 5.15).

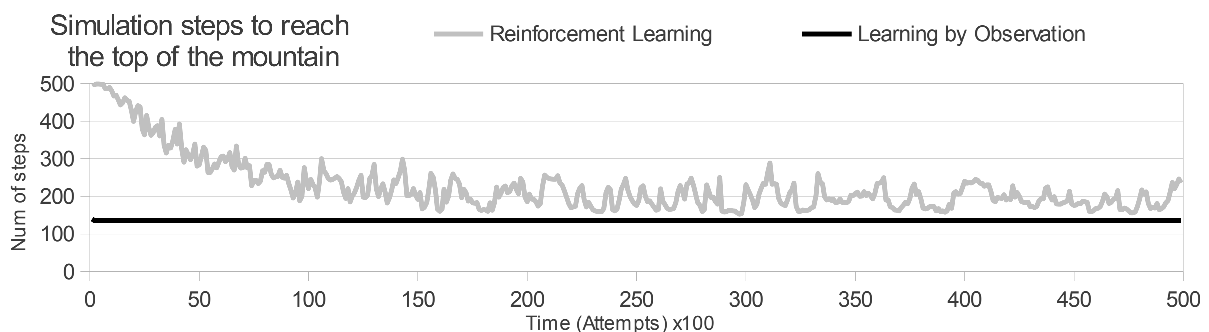


Figure 5.15: Results of experimenting the mountain car scenario - steps to reach the top

Figure 5.15 shows that the learning by observation agent requires less simulation steps (in comparison with the reinforcement learning apprentice) to go from the starting point to the top of the mountain after learning how to do it. The simulation steps spent in each attempt stabilizes at 136 (which is the same number steps spent by the expert) right after the agent has learnt the task of climbing the mountain. The learning by observation agent also requires less attempts to

learn how to reach the top (1 attempt as shown in table 5.15).

Figure 5.16 shows the progress of the expert and the two apprentice agents (learning by observation and reinforcement learning) throughout the experiment in terms of the distance travelled and the number of times reaching the top. Once again, for a clearer presentation, the results are combined in groups of 100 attempts to reach the goal.

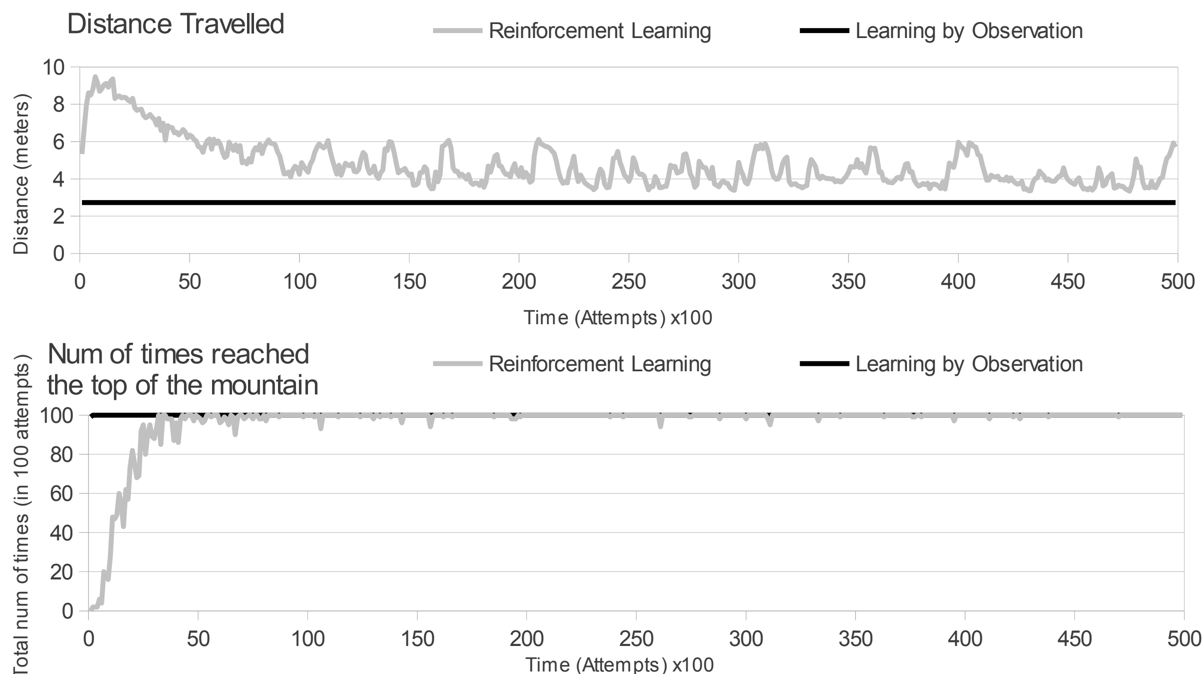


Figure 5.16: Results of experimenting the mountain car scenario - distance travelled and number of times reached the top

Figure 5.16 shows that the distance travelled by the learning by observation agent is smaller than the distance travelled by the reinforcement learning agent. The value of the distance, in the learning by observation agent, stabilizes at 2.73 meters (the same as the expert as table 5.15 shows) right after the agent learns the task of climbing the mountain, unlike what happens with the reinforcement learning agent. Even after a long learning period, the reinforcement learning agent is not able to learn the most efficient way to climb the mountain (the one that requires the least amount of actions and the smallest distance). Even the lowest values of the fluctuations of the number of actions and the travelled distance are still far away from those obtained by the learning by observation agent.

In addition, the reinforcement learning agent's capability of reaching the goal fluctuates between 90% and 100% of the times, until almost the end of the experiment. This means that even after learning how to reach the goal, the reinforcement learning agent is not always able to do it, something that is also reflected in the total number of times the agent reaches the top of

the mountain in table 5.15. In contrast, the learning by observation agent is always capable of reaching the top of the mountain after learning how to do it.

5.5 Validation of the Evaluation Criteria

The evaluation criteria, defined in section 1.3, can be validated from the outcome of the tests on the application scenarios. The first criterion indicates that agents must be capable of discovering experts from which they can potentially learn. The outcome of the tests confirms that the agents are capable of discovering and observing experts from which they learn how to perform a task. The tests on the software image (see section 5.1) also provide important information on the agent discovery and comparison processes.

The second criterion indicates that agents must be capable of solving a problem after observing an expert solving that same problem. The criterion is validated by testing the application scenarios in a setting where both the expert and the apprentice agents face the same conditions. The outcome of testing the application scenarios in this setting confirms that agents are capable of solving the same problems as the experts they observe.

The third criterion indicates that agents must be capable of solving a problem after observing an expert solving the same problem in different circumstances. The criterion is validated by testing the first and second application scenarios in a setting where the expert and the apprentice agents face different conditions. The outcome of testing the application scenarios in this setting confirms that agents are capable of solving a similar problem when facing different conditions.

The fourth criterion indicates that, after learning, the agent's performance, in terms of the number of actions correctly executed, must be similar to the performance of the observed experts. When both the expert and the apprentice agent face the same conditions, the apprentice is capable of executing the task with a performance that is identical to the performance of the experts it has observed. When facing different conditions, the agent is also capable of executing the task with a performance that is similar to the performance of the experts it has observed.

The fifth criterion indicates that agents must improve their ability to master a task after returning from an additional learning period. The outcome of the tests proves that when the apprentice goes through additional learning periods it increases its ability to perform a task because it performs less incorrect actions. This is more noticeable when the expert and the apprentice agent face different conditions.

The sixth criterion indicates that the learning by observation agent must obtain better results, in terms of the ability to master a task and the time it takes to learn that task, than other learning mechanisms under similar circumstances. The criterion is validated by the outcome of the tests on the second and third scenarios. The outcome of testing the second scenario shows that the learning by observation agent is the only one who performs the task correctly. Even though one of the agents executes the actions faster (the KStar agent), more than 50% of the task is incorrect (see section 5.3). The outcome of testing the third scenario shows that the learning by observation agent learns faster and performs the task more efficiently in terms of the distance travelled (see section 5.4).

Chapter 6

Conclusions and Future Work

The adoption of learning by observation solutions is relatively new in computer science and, as the latest approaches show (Sullivan, 2011; Kulic *et al.*, 2011; Tan, 2012; Fonooni *et al.*, 2012), it is still under development. With the exception of Machado's work (Machado, 2006; Machado & Botelho, 2006), the software approaches for learning by observation are commonly focused on state change information, completely ignoring the possibility of directly observing the agent performing the task. Even Machado's work has limitations since it only addressed the problem of learning vocabulary and it does not allow generalizations of the acquired knowledge. This means that, unlike the proposed approach, her apprentice agent is not able to learn control knowledge neither is it capable of dealing with conditions that are different from those faced by the observed experts.

Therefore, the proposed approach clearly contributes to advance the state of the art of learning by observation, providing software agents with a learning solution that is different from all other methods that are usually applied for software agents. Unlike the robotic approaches for learning by observation, which usually rely on physical sensors, the proposed software approach is not limited to software agents. It can also be used by robotic agents with minor adaptations, since all their physical actions are controlled by or reflected in software events. Even the data collected by robotic sensors needs a software representation (however complex it can be), since in its core the robot is effectively running a program and all high level decisions are made by that program.

Using the proposed approach, robotic agents could learn not only by observing actions in the tangible world but also by observing the interactions in the software world. For example, robot agents can learn how to use a remote control by observing software agents interacting

with the software API of the remote and also by observing someone pressing the buttons on the remote. A software approach for learning by observation provides a broader solution than approaches that are limited to the physical properties of robotic environments.

The experimental results in chapter 5 show that the proposed approach can be adapted to different domains. Learning by observation agents are capable of learning a task by observing an expert performing that same task and also by observing experts performing a similar task under different circumstances. The results show that the classification method is essential for situations in which the expert and the apprentice agent face different conditions and the recall method is essential for situations in which both the expert and the apprentice agent face the same conditions. The most usual scenarios for learning by observation, reported in the literature, address cases in which the apprentice and the expert agents face exactly the same conditions. However, in these cases the apprentice agent is only capable of learning the task performed by the expert. The inclusion of the classification method in the proposed approach extends the ability to learn by observation to other problems and domains.

The results also show that learning by observation agents are capable of determining when they need to acquire more knowledge and return to the learning state. The agent's evaluation enables learning by observation agents to enhance their knowledge even after they stop observing experts and start using the acquired knowledge, since after starting to use their knowledge they may go back to the learning state. This was one of the major drawbacks of previous learning approaches. The usual way of handling this is by manually feeding new examples whenever the agent requires them. In the case of the proposed approach, the agent is able to decide by itself when it needs to return to a learning state to observe experts. The process does not require any intervention since the agent can find the experts from which to collect the new training examples.

The outcome of the tests on the software image (see section 5.1) validates the proposed software image. The tests determine that the complexity of the agent's constituents can be measured by the number of parts, the number of elements in the part and depth of the parts. This complexity affects the time required to build and to compare the software image. The time required to observe the expert is not affected by this measure of the agent's complexity.

The outcome of the tests on the first scenario shows how the configurable parameters can influence the agent's performance and the importance of the software image meta-ontology. The results show that the agent is able to discover and correctly identify expert agents with

similar constituents and capabilities and also different experts containing the constituents and capabilities that are necessary for the task to learn. In both cases the agent was able to improve its knowledge on the task to learn. The outcome of the tests in section 5.2.5 also shows that it is possible to learn from an agent whose constituents have a different designation.

The outcome of the tests on the second and third scenarios shows that the proposed approach leads to better results than other learning mechanisms on similar circumstances. The second scenario shows that when some of the agent actions have no effects on the environment, observing only the effects of the actions is not enough for learning. Agents that only rely on observing the visible effects of agent actions were unable to learn how to perform the task correctly. The learning by observation agent was the only one who learnt how to perform the task both when facing the same conditions as the expert and when facing different conditions. Relying only on the effects of actions is even worst when the experts and the apprentice agents do not face the same conditions, which is usually the kind of situation where supervised learning algorithms are used.

The third scenario shows that the learning by observation agent is able to learn faster than the reinforcement learning agent in a situation where additional effort in developing expert knowledge is not necessary. The learning by observation is capable of reaching the goal for the first time faster than the reinforcement learning agent. It also does it more efficiently when considering the distance travelled and the number of actions required to reach the goal.

The only setback of the results of the tests on the second and third scenario is the time it takes for the learning by observation agent to propose and execute the actions. Even though the learning by observation agent is not the fastest to execute the actions, in the second scenario it is the only one who can execute the task correctly and in the third scenario it is the first to reach the goal. In addition, the performance of the learning by observation agent in the third scenario, in terms of the distance travelled and the number of actions, is approximately the same as the expert's performance.

Despite the contributions of the proposed approach, there are still problems left to solve, which means the approach can still be further improved. One of these problems is the necessity of an appropriate motivation mechanism that motivates the agent to learn new tasks regardless of the confidence in their knowledge. Other problems to solve consist of considering situations in which the observed experts are not executing the task to learn, where the conditions in observation hold the perspective of the observer and where agents can learn by observation and,

at the same time, execute actions. Future work may also consider opening the possibility of an agent learning how to perform a task that is radically different from the tasks performed by the observed experts.

The proposed software image can also provide important contributions to the software embodiment problem. Although the software image in this thesis is directed to learning by observation, its main purpose was to provide software agents with means to represent themselves on what can be called a body. As future work, the software image can continue its development beyond the purpose of learning by observation. Additional features of the software image can adapt it for the software embodiment and the grounding problems.

In the proposed approach the agent's motivation to learn is directly related to the confidence on its knowledge. The agent loses confidence, and returns to a state where it learns by observation, whenever it detects that it is executing incorrect actions or when it faces a certain amount of unfamiliar conditions. This limits the ability to learn new tasks because the agent only learns new things when it loses confidence on its knowledge.

The problem of the agent's motivation can be solved with an independent motivation mechanism. The mechanism could provide the agent with control over the decision to execute actions or learn by observation, establishing new rules for switching between these two states. The rules can include the confidence on the agent's knowledge, the tasks to learn, the agent's objectives and goals, which tasks are necessary to accomplish a goal and also the kind of experts available for observation. The motivation mechanism could also help the agent to focus observation on the experts that provide the knowledge for the task to learn. However, the solution requires additional processing, which depends on the amount of rules that are necessary to describe the agent's motivation. This eventually increases the time spent both when learning and when executing actions because the agent has to decide if it learns or executes actions in each iteration.

In the tested scenarios, all experts execute the actions that are necessary for the task to learn. However, this is not always true in real world situations. Even though the experts hold the knowledge on how to perform a task, they are not necessarily always executing that task. A possible solution for this problem consists of improving the communication between the expert and the apprentice. As future work, the feedback from the teachers could be enhanced and a communication protocol can be established between the apprentice agents and the teachers. The protocol allows apprentices to request the teacher to perform a specific task. The protocol can

also be used by the teachers to provide the corrections for the incorrect actions executed by the apprentices.

The solution for this problem only works when there are specialized experts that know how to communicate with the apprentices. This requires an additional effort in developing expert agents, something that is not required for the approach proposed in this thesis. The proposed approach does not require additional effort in developing expert knowledge because agents discover and observe the experts that already exist in the agent society. The functionalities of the software image gives the experts a passive role. They do not even need to know if they are being observed.

Another problem to solve is using the perspective of the apprentice agent in the observed conditions. In the proposed approach, the observed conditions hold the perspective of the expert agent because of the history record. Without the expert's perspective it is impossible to determine the conditions holding when the actions in the history record were executed. However, if the history record is disregarded, is it possible for an approach to acquire the conditions holding the apprentice's perspective and combine them with the actions in the snapshot.

The biggest obstacle of using the conditions holding the apprentice's perspective is the synchronization between the expert and the apprentice. The apprentice needs to know exactly when the expert acquires the conditions holding for the observed actions so that it can acquire the conditions at the same moment. A possible solution for this problem is enhancing the software image notifications and time-stamping the conditions in the snapshot. The timestamps point to the moment at which the expert acquired the conditions.

All agents should also have internal clocks that are synchronized each time the apprentice registers for notifications on new snapshots. The apprentice only needs to keep a record of the changes in the environment between the observed snapshots and use the timestamps to discover the correct conditions from this record. Even though this solution requires a simple synchronization it increases the amount of processing and storage space in the apprentice, which eventually increases the time spent learning.

Another innovation for the proposed approach is enabling the agent to learn and execute actions at the same time. This can be easily achieved by changing the possible activities of the learning and of the execution state. The agent would be able to discover and observe experts both in the learning and in the execution state. It would only need to be in the learning state when it has a low confidence on its knowledge and therefore needs to stop executing actions.

Although this is a simple improvement it does not guarantee an improvement in the agent's performance. It may even increase the time spent executing the actions because the agent has to share the resources with observation and the proposal and execution of actions.

Finally, future work may also consider the possibility of learning how to perform a task that is completely different from the tasks performed by the observed experts. With the proposed approach agents can only perform similar tasks when facing different conditions. To be able to perform a completely different task using only the information in its memory, the agent has to be able to rearrange it. The rearrangement can follow a specific logic or, as a simplification, it can be random. The agent rearranges its memory tree and executes actions according to this new arrangement. If it is capable of reaching a goal with this new arrangement, the path followed in the new arrangement is added to the memory.

This solution represents an additional method of learning for the proposed approach. Combining it with the agent's motivation mechanism may allow the motivation to indicate which of the two learning methods, learning by observation or learning from the agent's memory, are best suited for the current situation. Thereby, the agent's motivation not only indicates when the agent needs to learn but also which learning method it should use.

References

ALISSANDRAKIS, A., NEHANIV, C. & DAUTENHAHN, K. (2002). Imitation with alice: learning to imitate corresponding actions across dissimilar embodiments. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, **32**, 482–496. doi:10.1109/TSMCA.2002.804820.

ALPAYDIN, E. (2004). *Introduction to machine learning*. MIT Press, Cambridge Mass.

ARÉVALO, G., DUCASSE, S., GORDILLO, S. & NIERSTRASZ, O. (2010). Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. *Inf. Softw. Technol.*, **52**, 1167–1187. Available from: http://dx.doi.org/10.1016/j.infsof.2010.05.010, doi:10.1016/j.infsof.2010.05.010.

ARGALL, B.D., CHERNOVA, S., VELOSO, M. & BROWNING, B. (2009). A survey of robot learning from demonstration. *Robot. Auton. Syst.*, **57**, 469–483. Available from: http://dx.doi.org/10.1016/j.robot.2008.10.024, doi:10.1016/j.robot.2008.10.024.

BABES-VROMAN, M., MARIVATE, V.N., SUBRAMANIAN, K. & LITTMAN, M.L. (2011). Apprenticeship Learning About Multiple Intentions. In L. Getoor & T. Scheffer, eds., *ICML*, 897–904, Omnipress. Available from: http://dblp.uni-trier.de/db/conf/icml/icml2011.html#BabesMLS11.

BANDURA, A. (1977). *Social Learning Theory*. Prentice Hall, New York. Available from: http://books.google.pt/books?id=ehlxQgAACAAJ.

BEER, R.D. (1995). A dynamical systems perspective on agent-environment interaction. *Artif. Intell.*, **72**, 173–215. Available from: http://portal.acm.org/citation.cfm?id=201268.

BILLARD, A. (1999). Experiments in learning by imitation - Grounding and Use of Communication in Robotic Agents. *Adaptive Behavior*. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7685.

BILLARD, A. & HAYES, G. (1999). Drama, a connectionist architecture for control and learning in autonomous robots. *Adaptive Behavior*, **7**, 35—-64. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.8708.

BILLING, E., HELLSTROM, T. & JANLERT, L. (2010). Behavior recognition for learning from demonstration. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 866–872. doi:10.1109/ROBOT.2010.5509912.

BILLING, E., HELLSTRÖM, T. & JANLERT, L.E. (2011). Simultaneous control and recognition of demonstrated behavior. Tech. Rep. 15, UmeåUniversity, Department of Computing Science. Available from: http://umu.diva-portal.org/smash/record.jsf?pid=diva2:468094.

BOTELHO, L.M. & FIGUEIREDO, P. (2004). What your body and your living room tell my agent. In *Proceedings of the AAMAS 2004 Workshop "Balanced Perception and Action in Embodied Conversational Agents"*. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.81.3795.

BROOKS, R.A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, **2**, 14–23.

BROOKS, R.A. (1990). Elephants Don't Play Chess. *Robotics and Autonomous Systems*, **6**, 3–15. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.6594.

BROOKS, R.A. (1991). Intelligence Without Representation. *Artificial Intelligence*, **47**, 139—-159. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.1680.

BYRNE, R.W. (1999). Imitation without intentionality. Using string parsing to copy the organization of behaviour. *Animal Cognition*, **2**, 63–72. Available from: http://www.springerlink.com/content/pebwfdryvwtdnajr/, doi:10.1007/s100710050025.

BYRNE, R.W. & RUSSON, A.E. (1998). Learning by imitation: A hierarchical approach. *Behavioral and Brain Sciences*, **21**. Available from: http://journals.cambridge.org/production/action/cjoGetFulltext?fulltextid=30644, doi:10.1017/S0140525X98001745.

CALVOMERINO, B., GREZES, J., GLASER, D., PASSINGHAM, R. & HAGGARD, P. (2006). Seeing or Doing? Influence of Visual and Motor Familiarity in Action Observation. *Current Biology*, **16**, 1905–1910. Available from: http://eprints.ucl.ac.uk/2752/, doi:10.1016/j.cub.2006.07.065.

CEDERBORG, T., LI, M., BARANES, A. & YVES OUDEYER, P. (2010). P-y: Incremental local online gaussian mixture regression for imitation learning of multiple tasks. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*. Available from: http://hal.inria.fr/inria-00541778/.

CHERNOVA, S. (2009). *Confidence-based Robot Policy Learning from Demonstration*. Phd thesis, Carnegie Mellon University.

CHERNOVA, S. & VELOSO, M. (2009). Interactive policy learning through confidence-based autonomy. *Journal of Artificial Intelligence Research*, **34**, 1–25. Available from: http://dl.acm.org/citation.cfm?id=1622716.1622717.

CLEARY, J.G. & TRIGG, L.E. (1995). K*: An Instance-based Learner Using an Entropic Distance Measure. In *12th International Conference on Machine Learning*, 108–114, Morgan Kaufmann. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.4098.

CLEGG, B.A., DIGIROLAMO, G.J. & KEELE, S.W. (1998). Sequence learning. *Trends in Cognitive Sciences*, **2**, 275–281. Available from: http://dx.doi.org/10.1016/S1364-6613(98)01202-9, doi:10.1016/S1364-6613(98)01202-9.

COMITÉ, F. (2005). A Java Platform for Reinforcement Learning Experiments. *Journées Problèmes Décisionnels de Markov et Intelligence Artificielle PDMIA 2005*, 100–107. Available from: http://www.grappa.univ-lille3.fr/~ppreux/pdmia2005/pdmia_decomite.pdf.

CONSENS, M., MENDELZON, A. & RYMAN, A. (1992). Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering*, 138–156, ACM. Available from: http://dl.acm.org/citation.cfm?id=1925807, doi:10.1109/ICSE.1992.753496.

COSTA, E. & SIMÕES, A. (2008). *Inteligência Artificial - Fundamentos e Aplicações (2.ł Edição Revista e Aumentada)*. Tecnologias de Informação, FCA - Editora Informática.

COSTA, P.A.M. & BOTELHO, L.M. (2011). Software Image for Learning by Observation. In L. Antunes, H.S. Pinto, R. Prada & P. Trigo, eds., *Proceedings of the 15th Portuguese Conference on Artificial Intelligence*, 872–884, Lisbon, Portugal. Available from: http://iscte.pt/~luis/papers/EPIA2011_SoftwareImage.pdf, doi:ISBN:978-989-95618-4-7.

COSTA, P.A.M. & BOTELHO, L.M. (2012). Learning by Observation in Software Agents. *Proceedings of the 4th International Conference on Agents and Artificial Intelligence (ICAART 2012)*, **2**, 276–281. Available from: http://epia2011.appia.pt/LinkClick.aspx?fileticket=qa_T0FC_we4%3D&tabid=562.

COSTA, P.A.M. & BOTELHO, L.M. (2013). Learning by Observation of Agent Software Images. *Journal of Artificial Intelligence Research*, **47**, 313–349. Available from: http://www.jair.org/media/3989/live-3989-7119-jair.pdf.

DAUTENHAHN, K. (1994). Trying to imitate-a step towards releasing robots from social isolation. In *From Perception to Action Conference, 1994., Proceedings*, 290–301. doi:10.1109/FPA.1994.636112.

DAUTENHAHN, K. (1997). Biologically Inspired Robotic Experiments on Interaction and Dynamic Agent-Environment Couplings. In *In Proc. Workshop SOAVE'97, Selbstorganization von Adaptivem Verhalten*, 14—-24. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.1060.

DAUTENHAHN, K. & NEHANIV, C. (2002). The agent-based perspective on imitation. *Imitation in animals and artifacts*, 1–40. Available from: http://dl.acm.org/citation.cfm?id=762896.762898.

DEMIRIS, J. & HAYES, G. (2002). Imitation as a dual-route process featuring predictive and learning components: a biologically plausible computational model. In K. Dautenhahn & C. Nehaniv, eds., *Imitation in animals and artifacts*, 327–361, MIT Press, Cambridge.

DESCARTES, R. (1960). *Meditations on first philosophy*. Bobbs-Merrill. Available from: http://books.google.com/books?id=sxwXT7wLnW0C.

DI PELLEGRINO, G., FADIGA, L., FOGASSI, L., GALLESE, V. & RIZZOLATTI, G. (1992). Understanding motor events: A neurophysiological study. *Exp Brain Res*, **91**, 176–180.

DREYFUS, H.L. (1992). *What computers still can't do*. MIT Press. Available from: http://books.google.com/books?id=7vS2y-mQmpAC.

ESTEVES, A.S. & BOTELHO, L.M. (2007). The centrifugal development of artificial agents: a research agenda. In *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC '07, 977–982, Society for Computer Simulation International, San Diego, CA, USA. Available from: http://dl.acm.org/citation.cfm?id=1357910.1358063.

ETZIONI, O. (1993). Intelligence without Robots (A Reply to Brooks). *AI MAGAZINE*, **14**, 7—-13. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.8021.

FONOONI, B., HELLSTRÖM, T. & JANLERT, L.E. (2012). Learning High-Level Behaviors from Demonstration through Semantic Networks. *Proceedings of the 4th International Conference on Agents and Artificial Intelligence (ICAART)*. Available from: http://umu.diva-portal.org/smash/get/diva2:501519/FULLTEXT02.

FRANKLIN, S. (1997). Autonomous agents as embodied AI. *Cybernetics and Systems: An International Journal*, **28**, 499. Available from: http://www.informaworld.com/10.1080/019697297126029, doi:10.1080/019697297126029.

FRASCONI, P., GORI, M. & SODA, G. (1995). Recurrent neural networks and prior knowledge for sequence processing. *Knowledge Based Systems*, **8**, 313–332.

FREEMAN, E., FREEMAN, E., BATES, B., SIERRA, K. & ROBSON, E. (2004). *Head First Design Patterns*. O'Reilly Media. Available from: http://www.amazon.com/First-Design-Patterns-Elisabeth-Freeman/dp/0596007124.

GILES, C.L., HORNE, B.G. & LIN, T. (1995). Learning a class of large finite state machines with a recurrent neural network. *Neural Networks*, **8**, 1359–1365. Available from: http://clgiles.ist.psu.edu/papers/NN-1995-learning-FSMs.pdf.

HAJIMIRSADEGHI, H. & AHMADABADI, M. (2010). Conceptual imitation learning: An application to human-robot interaction. *Machine Learning*, 331–346. Available from: http://jmlr.csail.mit.edu/proceedings/papers/v13/hajimirsadeghi10a/hajimirsadeghi10a.pdf.

HALL, M., FRANK, E., HOLMES, G., PFAHRINGER, B., REUTEMANN, P. & WITTEN, I.H. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, **11**. Available from: http://www.cs.waikato.ac.nz/~ihw/papers/09-MH-EF-GH-BP-IHW-WEKAupdate.pdf.

HARNAD, S. (1990). The Symbol Grounding Problem. *Physica D*, 335–346. Available from: http://cogprints.org/615/1/The_Symbol_Grounding_Problem.html.

HEYES, C. & RAY, E. (2000). What is the significance of imitation in animals? *Advances in the Study of Behavior*, **29**, 215–245. Available from: http://www.sciencedirect.com/science/article/pii/S0065345408601060.

INAMURA, T., INABA, M. & INOUE, H. (1999). Acquisition of probabilistic behavior decision model based on the interactive teaching method. In *Ninth International Conference on Advanced Robotics, ICAR'99*.

ISAAC, A. & SAMMUT, C. (2003). Goal-directed learning to fly. In *Twentieth International Conference on Machine Learning*, 258—-265, AAAI Press. Available from: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.112.6146`.

JUDAH, K., FERN, A. & DIETTERICH, T. (2011). Active Imitation Learning via State Queries. In *ICML Workshop on Combining Learning Strategies to Reduce Label Cost*. Available from: `http://web.engr.oregonstate.edu/~judahk/pubs/imitationICML.pdf`.

KÖNIK, T. & LAIRD, J.E. (2006). Learning goal hierarchies from structured observations and expert annotations. *Machine Learning*, **64**, 263–287. Available from: `http://www.springerlink.com/content/p060623k16870325/http://www.springerlink.com/index/P060623K16870325.pdf`, doi: `10.1007/s10994-006-7734-8`.

KULIC, D., OTT, C., LEE, D., ISHIKAWA, J. & NAKAMURA, Y. (2011). Incremental learning of full body motion primitives and their sequencing through human motion observation. *The International Journal of Robotics Research*, **31**, 330–345. Available from: `http://ijr.sagepub.com/cgi/content/abstract/31/3/330`, doi: `10.1177/0278364911426178`.

KUSHMERICK, N. (1997). Software Agents and Their Bodies. *Minds and Machines*, **7**, 227—-247. Available from: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.6445`.

LAKOFF, G. & JOHNSON, M. (1980). *Metaphors we live by*. University of Chicago Press. Available from: `http://books.google.com/books?id=zX8DPcwhENgC`.

LOPES, M. & SANTOS-VICTOR, J. (2007). A developmental roadmap for learning by imitation in robots. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, **37**, 308–321. `doi:10.1109/TSMCB.2006.886949`.

LOPES, M., MELO, F. & MONTESANO, L. (2009). Active Learning for Reward Estimation in Inverse Reinforcement Learning. In *Eur. Conf. Machine Learning and Princ. Practice of Knowledge Disc. Databases*, 31–46. Available from: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.157.6655&rank=1`.

MACHADO, J.A. (2006). *Imagem Visual do Corpo de Software - Aquisição de Vocabulário por Observação*. Msc thesis, ISCTE.

MACHADO, J.A. & BOTELHO, L.M. (2006). Software agents that learn through observation (short paper). In *Proceedings of the International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. Available from: `http://iscte.pt/~luis/papers/AAMAS2006LearningShort.pdf`.

MAISTROS, G. & HAYES, G. (2001). An imitation mechanism for goal-directed actions. In *Proceedings of TIMR 2001 - Towards Intelligent Mobile Robots*. Available from: `http://www.dai.ed.ac.uk/homes/yuvalm/pubs/george-timr.ps.gz`.

MAISTROS, G. & HAYES, G. (2004). Towards an Imitation System for Learning Robots. In G. Vouros & T. Panayiotopoulos, eds., *Methods and Applications of Artificial Intelligence*, 246–255, Springer Berlin / Heidelberg. Available from: http://www.springerlink.com/content/4jx7kmqedt8x5qvj.

MATARIC, M.J. (1997). Studying the role of embodiment in cognition. In *In Cybernetics and Systems, Special issue on Epistemological Aspects of Embodied AI*, 457–470. Available from: http://www-robotics.usc.edu/personal/maja/publications/aaaifs96.ps.gz.

MELTZOFF, A.N. & MOORE, M.K. (1977). Imitation of facial and manual gestures by human neonates. *Science (New York, N.Y.)*, **198**, 75–8. Available from: http://www.sciencemag.org/content/198/4312/75.abstract, doi:10.1126/science.198.4312.75.

MEUNIER, M., MONFARDINI, E. & BOUSSAOUD, D. (2007). Learning by observation in rhesus monkeys. *Neurobiology of learning and memory*, **88**, 243–8. Available from: http://www.ncbi.nlm.nih.gov/pubmed/17572114, doi:10.1016/j.nlm.2007.04.015.

MITCHELL, T. (1997). *Machine learning*. McGraw-Hill Publishing Company, New York.

MOREL, B. & ACH., L. (2011). *Human Attention in Digital Environments*. Cambridge University Press. Available from: http://dx.doi.org/10.1017/CBO9780511974519.006.

NEAMTIU, I., FOSTER, J.S. & HICKS, M. (2005). Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, **30**, 1. Available from: http://dl.acm.org/citation.cfm?id=1082983.1083143, doi:10.1145/1082983.1083143.

NEHANIV, C. & DAUTENHAHN, K., eds. (2004). *Imitation and social learning in robots, humans, and animals: behavioural, social and communicative dimensions*. Cambridge University Press.

NEU, G. & SZEPESVARI, C. (2007). Apprenticeship learning using inverse reinforcement learning and gradient methods. In *23rd Conf. Uncertainty in Artificial Intelligence*, 295–302.

NEU, G. & SZEPESVÁRI, C. (2009). Training parsers by inverse reinforcement learning. *Machine Learning*, **77**, 303–337.

QUICK, T., DAUTENHAHN, K., NEHANIV, C.L. & ROBERTS, G. (2000). The essence of embodiment: A framework for understanding and exploiting structural coupling between system and environment. *AIP Conference Proceedings*, **517**, 649–660. Available from: http://link.aip.org/link/?APC/517/649/1, doi:10.1063/1.1291299.

RAMACHANDRAN, D. & AMIR, E. (2007). Bayesian inverse reinforcement learning. *Urbana*, **51**, 61801. Available from: http://www.aaai.org/Papers/IJCAI/2007/IJCAI07-416.pdf.

RAMACHANDRAN, V. (2000). Mirror neurons and imitation learning as the driving force behind "the great leap forward" in human evolution. *Edge*, **69**. Available from: http://www.edge.org/3rd_culture/ramachandran/ramachandran_index.html.

RAMACHANDRAN, V. (2003). *The emerging mind: the BBC Reith Lectures 2003*. Profile Books. Available from: http://books.google.com/books?id=u4JuQgAACAAJ.

RAO, R.P.N., SHON, A.P. & MELTZOFF, A.N. (2004). A bayesian model of imitation in infants and robots. In *Imitation and Social Learning in Robots, Humans, and Animals*, 217–247, Cambridge University Press.

RIZZOLATTI, G., FADIGA, L. & GALLESE, V. (1996). Premotor cortex and the recognition of motor actions. *Cognitive brain research*, **3**, 131–141. Available from: http://www.ncbi.nlm.nih.gov/pubmed/8713554http://www.sciencedirect.com/science/article/pii/0926641095000380.

RIZZOLATTI, G., FOGASSI, L. & GALLESE, V. (2000). Cortical mechanisms subserving object grasping and action recognition: A new view on the cortical motor functions. *MIT Press*, 539–552.

RUSSELL, S. (1998). Learning agents for uncertain environments (extended abstract). In *Proceedings of the eleventh annual conference on Computational learning theory*, COLT' 98, 101–103, ACM, New York, NY, USA. Available from: http://doi.acm.org/10.1145/279943.279964, doi:10.1145/279943.279964.

SAMMUT, C., HURST, S., KEDZIER, D. & MICHIE, D. (1992). Learning to fly. In *In Proceedings of the Ninth International Conference on Machine Learning*, 385—-393, Morgan Kaufmann. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.3725.

SAVAGE, T. (2003). The grounding of motivation in artificial animals: Indices of motivational behavior. *Cogn. Syst. Res.*, **4**, 23–55. Available from: http://dx.doi.org/10.1016/S1389-0417(02)00070-0, doi:10.1016/S1389-0417(02)00070-0.

SUGIMOTO, N., MORIMOTO, J., HYON, S.H. & KAWATO, M. (2012). The eMOSAIC model for humanoid robot control. *Neural networks : the official journal of the International Neural Network Society*, **29-30**, 8–19. Available from: http://www.ncbi.nlm.nih.gov/pubmed/22366503, doi:10.1016/j.neunet.2012.01.002.

SULLIVAN, K. (2011). Multiagent hierarchical learning from demonstration. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, 2852–2853, AAAI Press. Available from: http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-498, doi:10.5591/978-1-57735-516-8/IJCAI11-498.

SUN, R. (2001). Introduction to sequence learning. In R. Sun & C.L. Giles, eds., *Sequence Learning - Paradigms, Algorithms, and Applications*, vol. 1828 of *Lecture Notes in Computer Science*, 1–10, Springer. doi:http://link.springer.de/link/service/series/0558/bibs/1828/18280001.htm.

SUTTON, R.S. & BARTO, A.G. (1998). *Reinforcement learning*. MIT Press, Cambridge, MA. Available from: http://mitpress.mit.edu/0262193981.

TAN, H. (2012). Implementation of a Framework for Imitation Learning on a Humanoid Robot Using a Cognitive Architecture. In D.R. Zaier, ed., *The Future of Humanoid Robots - Research and Applications*, chap. 10, InTech. Available from: http:

//www.intechopen.com/source/pdfs/25776/InTech-Implementation_of_a_framework_for_imitation_learning_on_a_humanoid_robot_using_a_cognitive_architecture.pdf.

VEERARAGHAVAN, H. & VELOSO, M. (2008). Teaching sequential tasks with repetition through demonstration. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 3*, AAMAS '08, 1357–1360, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC. Available from: http://dl.acm.org/citation.cfm?id=1402821.1402871.

WOOD, M.A. (2008). *An agent-independent task learning framework*. Phd thesis, University of Bath. Available from: http://opus.bath.ac.uk/14407/.

ZIEMKE, T. (2003). What's that thing called embodiment. In Alterman & Kirsh, eds., *Proceedings of the 25th Annual meeting of the Cognitive Science Society*, 1134–1139, Lawrence Erlbaum, Mahwah, NJ. Available from: http://csjarchive.cogsci.rpi.edu/proceedings/2003/pdfs/244.pdf.

A P P E N D I C E S

Appendix A

Embodiment and Embodied Cognition

This section describes what is embodiment and how embodiment evolved throughout history. It provides an insight on the importance of the role of the body and shows how embodiment can be useful to solve the grounding problem.

Embodiment consists of giving the notion of body to an entity. It is supported by the phenomenology school, which believes that the role of the body, in the interaction between agent and environment, should be more significant. Agent involvement in the world cannot be understood without the existence of a body, thus embodiment is a necessity (Kushmerick, 1997). Under the phenomenological perspective, instead of characterizing the body, at best, as an insensitive support and input/output system for the generative apparatus of the mind, the entire human agent is seen as a material system that is born, develops and reasons through material action and interaction in a material environment (Quick *et al.*, 2000).

A.1 Embodiment Through History

The significance of the role of the body became the ground for strong criticism to classical AI. Within this criticism a new paradigm, the nouvelle AI, was born. Nouvelle AI holds a phenomenological view of how body and intelligence are related. Classical AI is supported by the traditional cognitivist school, which believes that cognition is purely a symbolic manipulation of mental representations. Under the cognitivist view, embodiment and the body are assigned to peripheral roles (Quick *et al.*, 2000). According to Plato and Decartes, major supporters of the cognitivist school, the body was getting in the way of intelligence, rather than being indispensable for it (Descartes, 1960).

On the other hand, the nouvelle AI view claims that agent involvement in the world cannot be understood only through mental and symbolical representations. An agent can only make sense of its surroundings if the world is described as the agent perceives it, from its sensors, and as the agent interacts with it, through its actions (Brooks, 1990, 1991). For a full understanding of the agent's involvement in the world, the agent's body, that is, its constituents and capabilities, must also be taken into account (Kushmerick, 1997).

According to Dreyfus (Dreyfus, 1992), phenomenological observations allow shifting the need of proof to those who give priority to mental representations. Although these observations are more illustrative than conclusive, they reveal that humans do not need mental states in their everyday activities. Activities such as opening a door, or simply walking do not need internal symbolic representations in our minds. We can do them easily without thinking too much on them.

The need of a more relevant role for the body is also unveiled by Lakoff and Johnson (Lakoff & Johnson, 1980) on their psycholinguistic analysis of metaphorical projections. Under their perspective, our ability to understand and reason abstractly derives from our bodily experience rather than the other way around. This theory is based on their analysis of metaphorical projections, such as the use of physical properties when one refers to an argument. The argument 'stands' or 'falls' according to its foundation, which can have a 'solid support' or a 'week support'.

Lakoff and Johnson observed that bodily movement, manipulation of objects and perceptual interactions involve recurring patterns, which they call the image schemata. The metaphorical projection is explained as the transformation of the image schemata into new situations. Taking the previous example, the original image schemata may have come from a regular object that can fall or stand, depending on the structure of its foundation. By metaphorical projection this image schemata was transformed from the observable object to the abstract term 'argument'. The relation between metaphor and embodiment is due to its origin from a bodily experience. This claim is based on empirical evidence since there are several examples of non-physical structures being treated as physical structures. The same does not happen in the other direction (Lakoff & Johnson, 1980).

Besides the support by empirical evidence, the image schemata encodes patterns of activity in ways that are well suited for the sorts of transformations that metaphorical projection requires. This means that image schemata are encoded in a way that efficiently renders the

natural transformations needed for metaphorical projection.

Both phenomenological and psycholinguistic observations defend that even high cognitive functions such as reasoning and understanding require the notion of body. Even though mental representations are also required by these high cognitive functions, we still need to orient ourselves in a familiar world in order to act deliberately (Lakoff & Johnson, 1980; Dreyfus, 1992).

A.2 The Grounding Problem

If software agents had a notion of their bodies they would be able to overcome some of the difficulties they face when solving traditional AI problems. Many of those are related to the fact that, for too long, AI systems have simply manipulated ungrounded meaningless internal symbols. Standard AI theories of action reduce moving one block on top of another to symbolic structures such as STACK(BLOCK-A, BLOCK-B). Since these symbols have no first-person meaning for AI systems, there is no reliable way to create internal mappings between those symbols and the real world. This observation led Harnad (Harnad, 1990) to be faced with the symbol grounding problem.

The symbol grounding problem is not limited to symbolic representations and it comprehends more than simply symbol grounding. Savage (Savage, 2003) includes to this problem all the transactions the agent makes with the world. Such transactions need to have an intrinsic meaning that cannot result exclusively from assumptions and predispositions made by the designer.

For nouvelle AI, the development of cognitive and intelligent systems depends on overcoming the grounding problem (Quick *et al.*, 2000; Esteves & Botelho, 2007). To do it, agents need to be linked to the environment in the same way as organisms are. Only with the embedding of the agent in the real word can the grounding problem be solved.”Without an ongoing participation and perception of the world there is no meaning for an agent” (Brooks, 1991). Embodiment emerges from this direct interplay between agent and environment. Through embodiment, agents can dwell in the world in such a way as to avoid the task of formalizing everything because the body enables it to bypass this formal analysis (Quick *et al.*, 2000; Dreyfus, 1992; Brooks, 1990; Esteves & Botelho, 2007).

Several authors managed to create environment embedded robot architectures (behaviour

based robots) that are able to directly map sensor inputs to actuators (Brooks, 1990; Dautenhahn, 1997; Beer, 1995). Robots built in this way prove that there is no need for symbolic models of the world, neither any kind of symbolic processing, discarding the need for symbol grounding.

These robot architectures also prove that it is possible to carry out open-ended tasks in unconstrained environments while at the same time remaining simple. Solving this same problem through classical AI requires an increase in agent complexity. Classical AI symbol systems tend to increase their complexity as their beliefs grow. This is a necessity for symbol systems, since they need objectivity (Dreyfus, 1992; Brooks, 1990; Beer, 1995; Quick *et al.*, 2000; Botelho & Figueiredo, 2004).

The classical AI solution would also be limited to isolated cognitive skills in restricted task domains. This makes this solution ever more biologically implausible, proving that the symbol system used by classical AI is an inadequate path to strong AI (Brooks, 1990; Dreyfus, 1992; Beer, 1995).

A.3 Computational Approaches for Embodiment

Defining what is embodiment and devising a way to achieve it are some of the questions posed when embracing a computational approach for embodiment. Despite several proposals on computational embodiment (Quick *et al.*, 2000; Kushmerick, 1997; Beer, 1995; Franklin, 1997; Brooks, 1990; Etzioni, 1993), there are still no full implementations of embodied agents, especially on software agents because of the need for a physical body (Ziemke, 2003; Quick *et al.*, 2000). As Ziemke (Ziemke, 2003) noticed, one of the main reasons for this is the lack of a common understanding on what exactly is embodiment, what kind of body, if any, is necessary for embodiment and how it can be implemented.

Computer programs can sometimes be considered the epitome of abstract, detached and disembodied reasoning. This is one of the reasons why it can be difficult to understand how to embody something in a software world, given that claims about embodiment revolve around materiality and the physical world (Quick *et al.*, 2000). Dreyfus (Dreyfus, 1992) suggests that embodiment needs relatively little emphasis on the actual form of the body, but rather on qualitative features such as the capacity to act, *"the 'I can' ability to respond to situational solicitations"* (Dreyfus, 1992).

The possibility of performing actions is not something that is inherited from being material,

unless the action is happening in the material world. This makes being a material agent not immediately significant (Quick *et al.*, 2000). Therefore, stripping embodiment from its association to physicality makes it possible to find applications of it for software agents. One of these possible applications is learning by observation, as stated by Botelho and Figueiredo (Botelho & Figueiredo, 2004) and Mataric (Mataric, 1997).

Etzioni (Etzioni, 1993) was one of the first authors that realized the importance of embodiment in software environments. He introduced the *softbots* paradigm as the application of Brooks (Brooks, 1990) tangible world robots on software environments such as operating systems and databases. In his perspective, agents should be able to interact with these environments using commands and interpreting the responses to those commands.

For Etzioni, using *softbots* is preferred to using mobile robots, since the first are easier to build, provide an easier embodiment and their environment is subject to less noise. Etzioni also highlights the importance of providing software agents with sensing abilities. For him, a software agent can be situated in the software environment in the same way as a robot is situated in the physical world, if it is characterized by what it can do (its actions) and also by the kind of inputs it is able to collect from the software environment (Etzioni, 1993).

Etzioni's ideas were followed by Kushmerick (Kushmerick, 1997) and Franklin (Franklin, 1997), who also tried import benefits of embodiment to the software agent world by putting aside the materiality clause of embodiment. Kushmerick's solution for embodiment consists on broadcasting environmental information and making agents respond to it in an interrupt-driven manner because, in the physical world, the environment information is constantly broadcasted to agents and when the agent sensors receive it they immediately send it to the responsible higher cognitive mechanisms (Kushmerick, 1997)

This solution has, however, some other problems such as the costs associated with the high-bandwidth coupling, and the large amounts of information that need to be processed by the agent. A possible solution for handling this large amount of information is throwing most of it out. Various high bandwidth sensors that humans and animals have, such as the eyes with foveation, do this. Eyes have a relatively small area of high-resolution capture, the fovea, surrounded by a much larger area of low-resolution sensors. Another possibility is using attention mechanisms which may help the agent to focus its sensors on a particular event. When something relevant is happening, or something suddenly changes its state, the attention is immediately caught by that event (Kushmerick, 1997).

Recent approaches for software embodiment are usually associated to the representation of virtual characters, such as described in (Morel & Ach., 2011). Like the robotic approaches, the problem of the existence of a body is overcome by replacing it with a virtual representation of something that exists in the physical world, which in this case is the body of the virtual character.

Appendix B

Description of The Software Image Ontology

This appendix presents the owl specification of the software image meta-ontology and of the ontologies used in the examples of section 3.3. The software image meta-ontology defines the basic elements and the possible relationships between those elements for the ontologies used to represent the agent sensors, visible attributes, actions and tasks. Figure B.1 shows the OWL representation of the software image meta-ontology presented in figure 3.6.

Any ontology that represents the designations of the tasks and atomic elements of the software image must follow the specification described in figure B.1. Figure B.2 shows the OWL specification of the example provided in section 3.3 (see figure 3.7 and 3.8).

The OWL specification in figure B.2 shows an example of how the *equivalentTo* relationship can be used. The relationship allows the *dateAsObj* sensor to be regarded as corresponding to the *dateAsStr* sensor. A conversion class, the *DateConverter* converts the information provided by one sensor to the information provided by the other.

```
<owl:Class rdf:ID="Task"/>
<owl:Class rdf:ID="Atomic"/>
<owl:Class rdf:ID="Action">
    <rdfs:subClassOf rdf:resource="#Atomic" />
</owl:Class>
<owl:Class rdf:ID="DataSource">
    <rdfs:subClassOf rdf:resource="#Atomic" />
</owl:Class>
<owl:Class rdf:ID="Sensor">
    <rdfs:subClassOf rdf:resource="#DataSource" />
</owl:Class>
<owl:Class rdf:ID="VisibleAttribute">
    <rdfs:subClassOf rdf:resource="#DataSource" />
</owl:Class>
<owl:Class rdf:ID="EquivalentTo">
    <rdfs:subClassOf>
        <owl:Restriction>
        <owl:onProperty rdf:resource="#translation"/>
        <owl:maxCardinality rdf:datatype="&xsd;
          nonNegativeInteger">
            1
        </owl:maxCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
<owl:ObjectProperty rdf:ID="performsTask">
    <rdfs:domain rdf:resource="#Atomic" />
    <rdfs:range  rdf:resource="#Task" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="equivalentTo">
    <rdfs:domain rdf:resource="#Atomic" />
    <rdfs:range  rdf:resource="#EquivalentTo" />
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="translation">
    <rdfs:domain rdf:resource="#EquivalentTo" />
    <rdfs:range  rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="datatype">
    <rdfs:domain rdf:resource="#DataSource" />
    <rdfs:range  rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

Figure B.1: The OWL specification of the software image meta-ontology

```
%Agent A’s sensor
<Sensor rdf:ID=”dateAsObj”>
    <datatype rdf:datatype=”&xsd;string”>Date</datatype>
    <equivalentTo rdf:resource=”#DateObjStr”/>
</Sensor>
%Agent B’s sensor
<Sensor rdf:ID=”dateAsStr”>
    <datatype rdf:datatype=”&xsd;string”>String</datatype>
    <equivalentTo rdf:resource=”#DateObjStr”/>
</Sensor>
%The equivalentTo property
<EquivalentTo rdf:ID=”DateObjStr”>
    <translation rdf:datatype=”&xsd;string”>
       DateConverter
    </translation>
</EquivalentTo>
```

Figure B.2: An example of an agent ontology in OWL

Appendix C

Testing Machine Learning Algorithms

This section describes the experiment that tests several machine learning algorithms in learning by observation environments. The analysis of these algorithms gave rise to the classification method of learning described in section 4.5.2. The experiment shows what types of algorithms are best suited for this learning approach and how the different settings affect the algorithm's performance.

A test framework was developed to allow the creation of small test scenarios consisting of one expert that provides examples and one apprentice agent using different learning algorithms. In this framework, the state of the environment is described by four variables with a limited set of possible values: two numerical, one string and one enumerate. The expert agent performs actions when the state of the environment reflects specific combinations of the four environment variables.

The unit of time in the test scenarios is the simulation step. In each simulation step the expert acquires the current state of the environment and performs a sequence of actions. The apprentice agent acquires a training example consisting of the state of the environment before the expert actions were executed and the actions executed by the expert.

After acquiring the training example, the apprentice faces a sequence of twenty different states of the environment and uses its learning algorithm to propose actions for each state in the sequence. An external evaluator observes the actions proposed by the apprentice for those twenty different states and determines how many actions were appropriate.

Table C.1 presents the settings of four tested scenarios. The settings show how facing the same problem or a different problem influences the efficiency of the learning algorithm in terms of the number of correct actions proposed. When the agent faces the same problem as the

observed expert, the states of the environment in the sequence faced by the apprentice (in each simulation step) are also faced by the expert. When this happens, after twenty simulation steps the agent has observed all the possible states.

Table C.1: The settings for the test scenarios.

Setting	1	2	3	4
Is facing the same problem	yes	no	yes	no
Can observe the whole environment	yes	yes	no	no

On the other hand, when the agent faces a different problem, the states of the environment in the sequence faced by the apprentice are different from those observed on the expert. In this case, the agent has to rely on the learning algorithm to choose the examples that most closely resemble the faced states of the environment.

The settings in table C.1 also show how the algorithm behaves when the agent cannot acquire all the variables that describe the state of the environment. These third and fourth setting simulate a situation where the effects of some actions are not visible in the state of the environment. They show how machine learning algorithms that can only observe the effects of the actions in the environment behave in these situations.

The experiment tested six different algorithms in the four different settings. The tested algorithms faced exactly the same circumstances. The tested algorithms are implementations of the conjuntive rule, Multi-Layer Perceptron back-propagation (MLP), KStar, Nearest Neighbour like algorithm using non-nested Generalized Exemplars (NNGE), ID3 and Naive Bayes, provided by the WEKA (Waikato Environment for Knowledge Analysis) software workbench (Hall *et al.*, 2009). Each test lasted thirty simulation steps, meaning the apprentice had the chance to observe the expert thirty times. Figure C.1 shows how the performance of each algorithm, in terms of being able to propose the correct actions, evolved throughout the test for the first and second settings.

Figure C.1 shows that all algorithms performed better on the first setting. Even though the MLP proposes all the actions correctly after observing the expert only fourteen times, its performance is not stable throughout the simulation. From that point forward, the rate of correct actions changes between being all correct and only 95% correct. On the other hand, all the actions proposed by the NNGE, the ID3 and the KStar algorithm are correct after observing the expert twenty times, which means the agent was able to learn the task. However, the KStar outperforms these three algorithms because it holds the highest rate of correct actions before

the twenty observations.

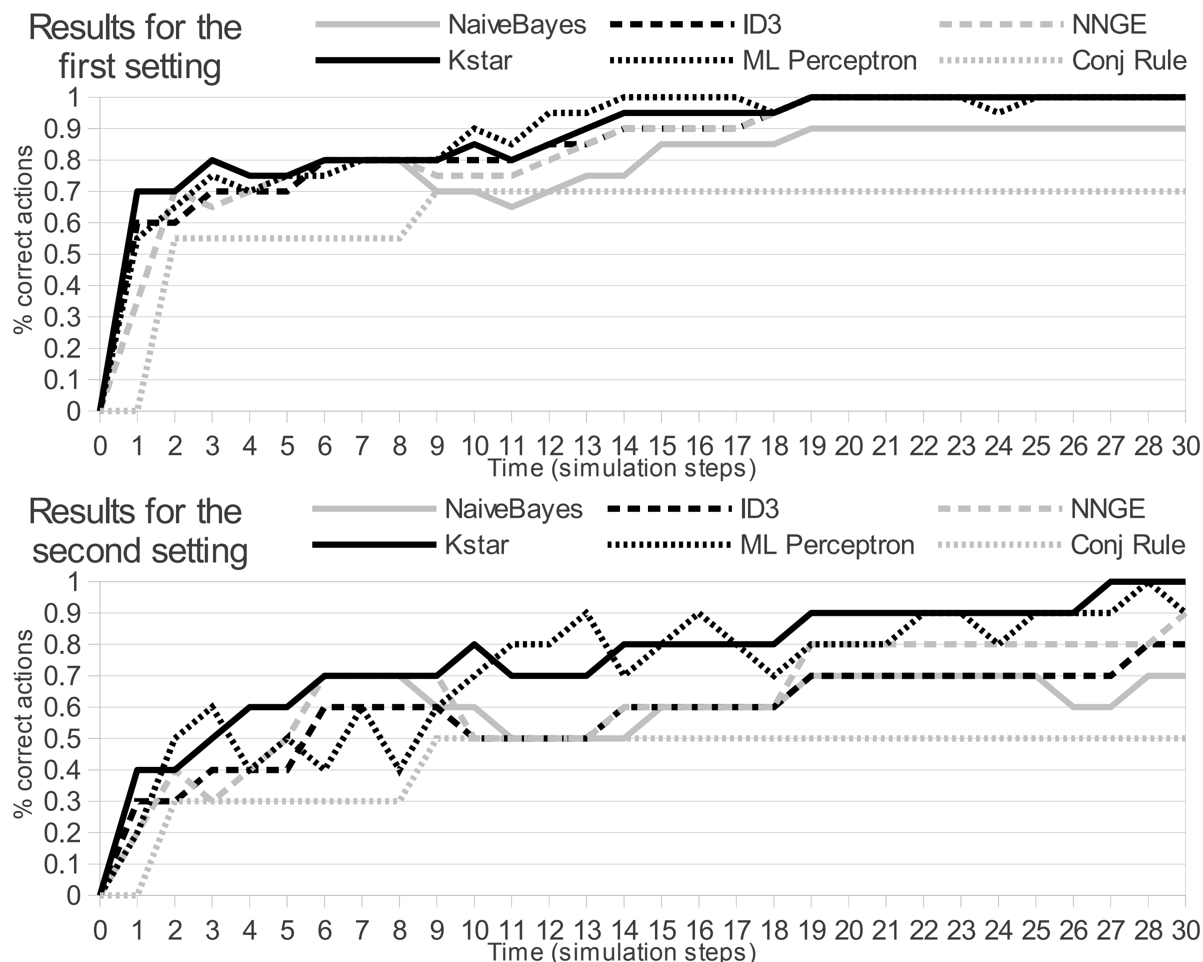


Figure C.1: The results for first and second settings

On the second setting the KStar and the MLP have the best performance, even though they take more time for all the proposed actions to be correct (about twenty seven simulation steps). Once again the performance of the MLP does not stabilize after being capable of proposing all actions correctly, which is not the case for the KStar algorithm. As for the remaining algorithms, they were not capable of proposing all actions correctly until the end of the simulation.

Further research on this matter revealed that the algorithm's confidence on the proposed actions was quite low, meaning the algorithm's decision is prone to errors. It would be possible for some of these algorithms (such as the NNGE and the ID3) to achieve this goal if the simulation lasted longer. However, the data collected from running the experiment in these two settings is sufficient for determining that the KStar algorithm is a good choice for learning by observation. Figure C.2 shows the test results for the third and fourth setting.

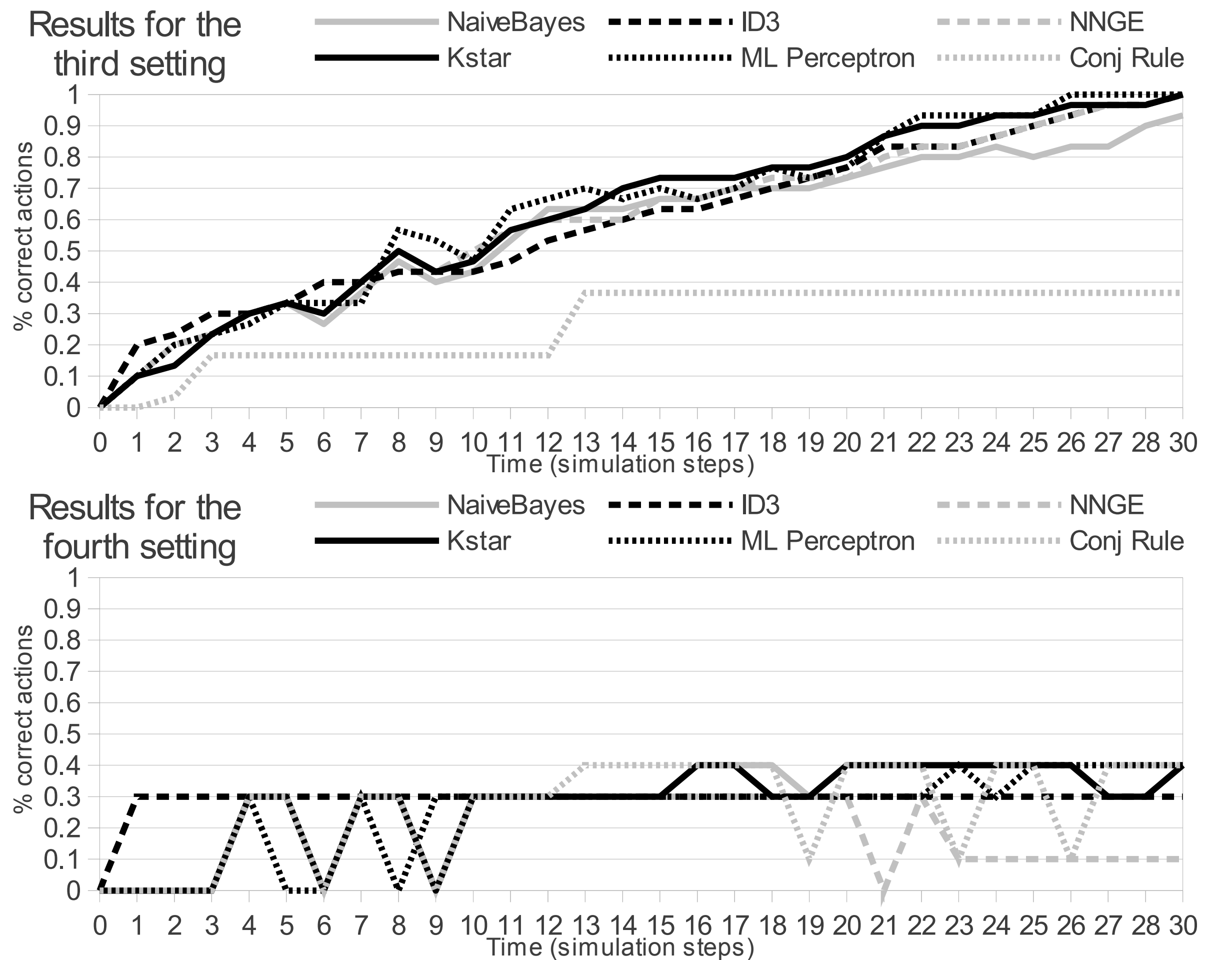


Figure C.2: The results for third and fourth settings

Figure C.2 shows that, once again, the algorithms performed better when facing the same problems as the observed expert (the third setting), even though the results are worse than on the first two settings. This is a clear indicator that observing only the effects of actions when some of those actions have no visible effects in the environment hinders learning. With the exception of the Conjuntive Rule algorithm, all the algorithms have similar performances. The ML perceptron was the only algorithm capable of proposing all actions correctly in the time the simulation takes (26 simulation steps). However, the remaining algorithms, with the exception of the Conjuntive Rule, could easily achieve that goal if the simulation took longer.

The same does not happen in the fourth setting, as figure C.2 shows. Only 40% or less of the actions proposed by the algorithms are correct when the apprentice and the expert face different problems. The agent has no way of knowing what actions to propose using the observed examples because it is not possible to see all the effects of the actions. When the expert and the apprentice do not face the same transition of states, it becomes impossible for the apprentice

agent to know exactly what actions to execute. Therefore, relying only on the changes in the environment leads to ineffective learning.

Although only a small set of training examples was used for these experiments, the first two settings show that the agent is able to learn almost all the expert's actions in a small amount of time (see figure C.1). When compared with other learning techniques that need longer training periods such as reinforcement learning, results show that these learning algorithms allow software agents to achieve results faster.

The results also show that the KStar and the MLP algorithms are those from which it is possible to learn from fewer examples. In addition to the ability to learn from few examples, the algorithms must also be efficient in the time it takes to learn from the examples and propose actions. Table C.2 shows the average time taken by the tested algorithms to learn from the examples and propose actions.

Algorithm	N. Bayes	ID3	NNGE	KStar	ML Perceptron	Conj. Rule
Time (ms)	3.09	1.18	0.94	0.47	118.03	0.58
St. Deviation	1.16	0.48	0.61	0.33	85.43	0.33

Table C.2: The average time taken to learn from the examples and propose actions

Table C.2 shows that the KStar is the fastest algorithm to learn and execute the task and the MLP is the slowest. Even though the MLP learns faster it takes longer to execute the task. The combination of the results from figure C.1 and table C.2 reveals that KStar is the most suited algorithm for the classification method of the proposed approach.

Appendix D

Scenario Settings

This appendix describes the distribution of the elements that participate in the application scenarios. It shows the UML component diagrams of the scenario elements and the class diagrams of the participant agents. The implementation of the scenarios and their source code is accessible from a a software versioning and revision control repository [1].

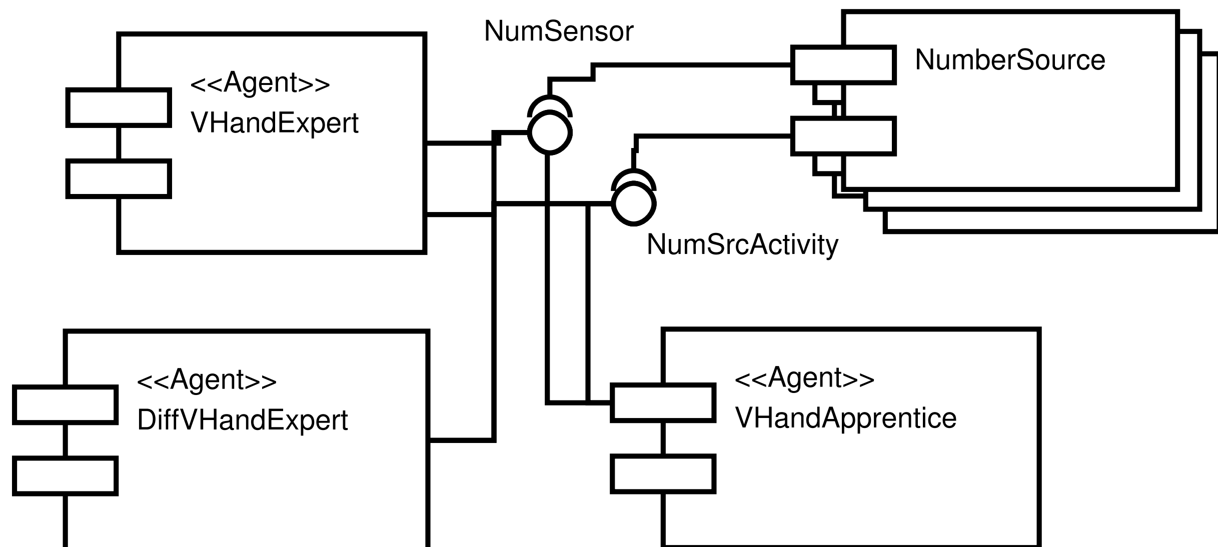


Figure D.1: The component diagram of the elements of the virtual hand scenario

[1] https://github.com/coostax/lckosa-phd.git

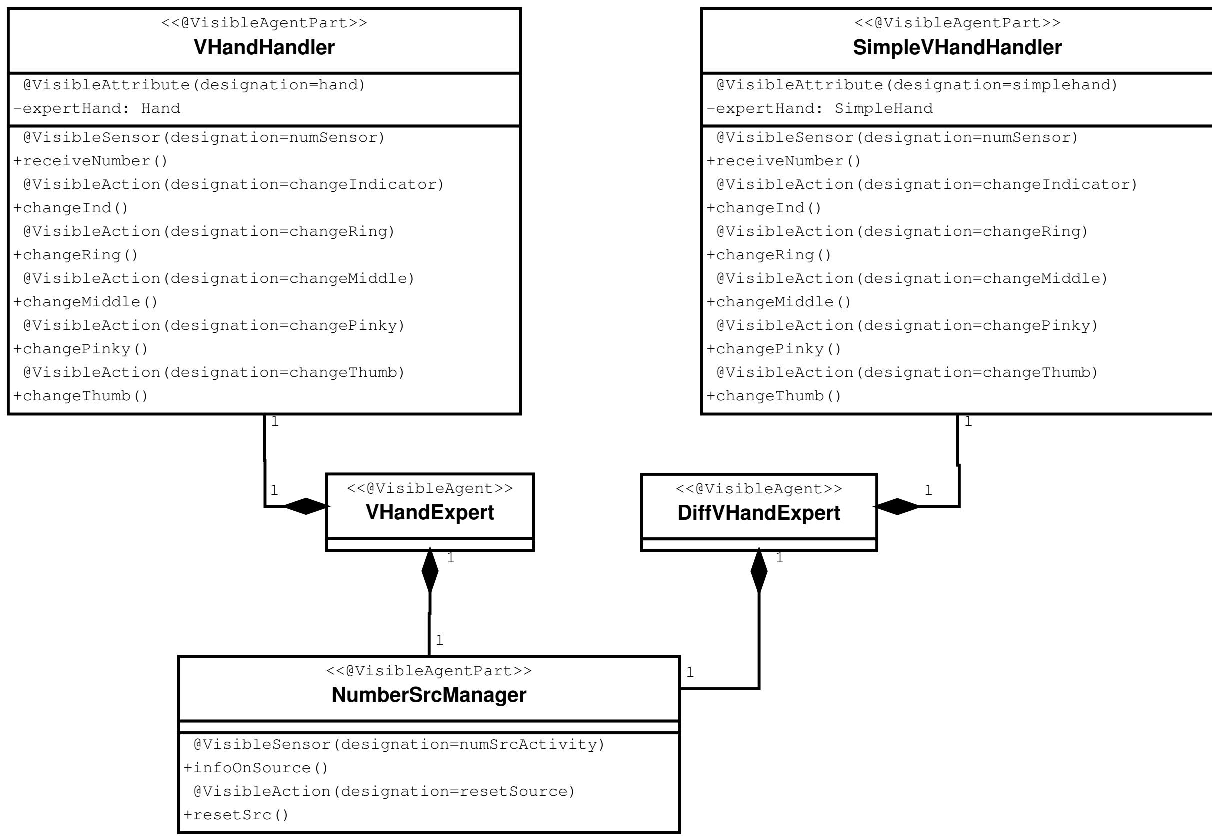


Figure D.2: The class diagram of the two kinds of expert agents in the virtual hand scenario

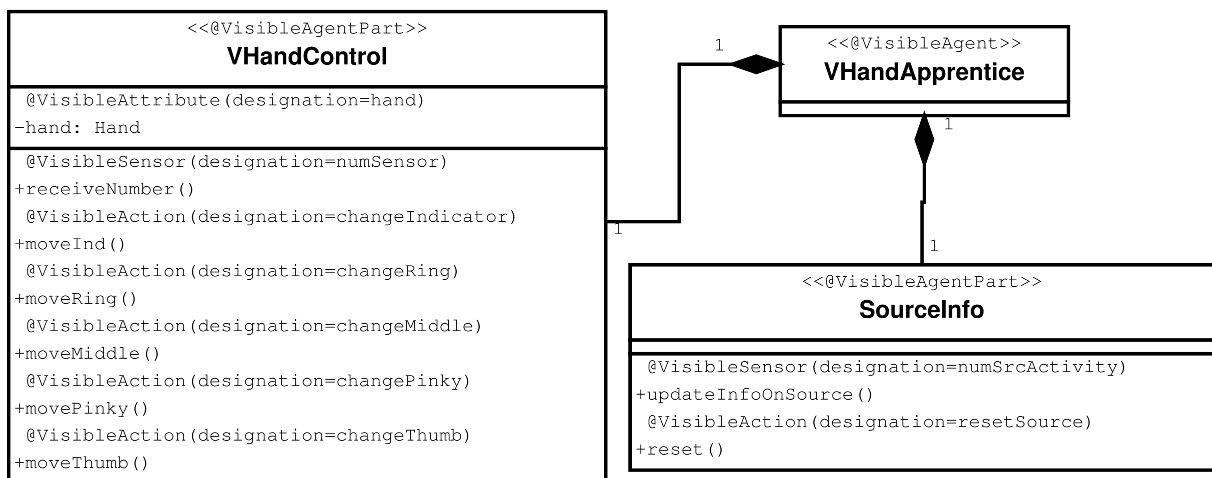


Figure D.3: The class diagram of the apprentice agent in the virtual hand scenario

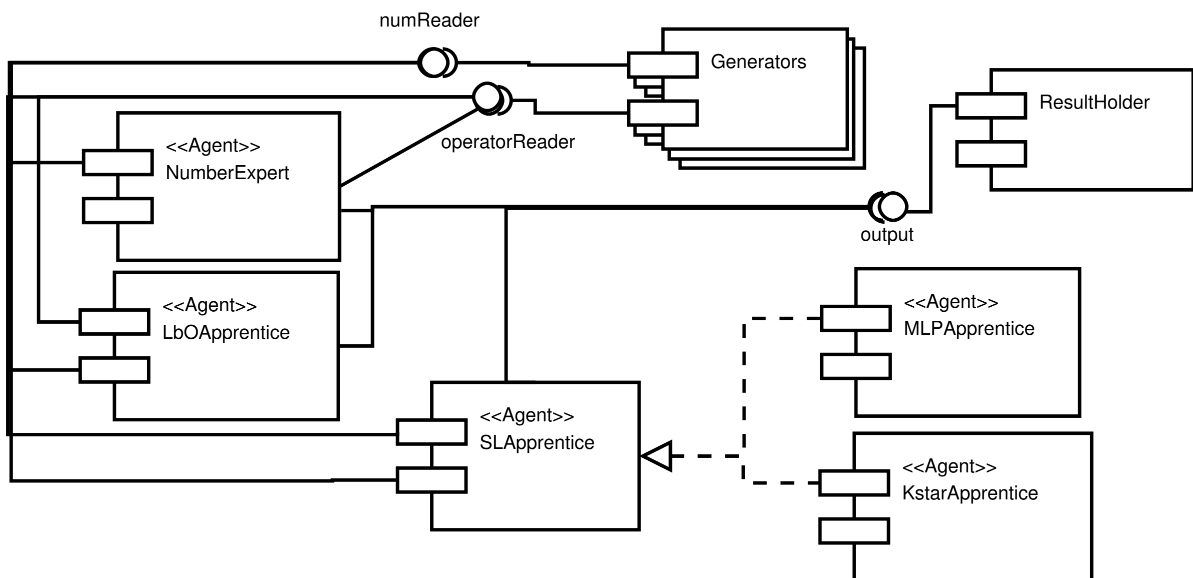


Figure D.4: The component diagram of the elements of the number calculator scenario

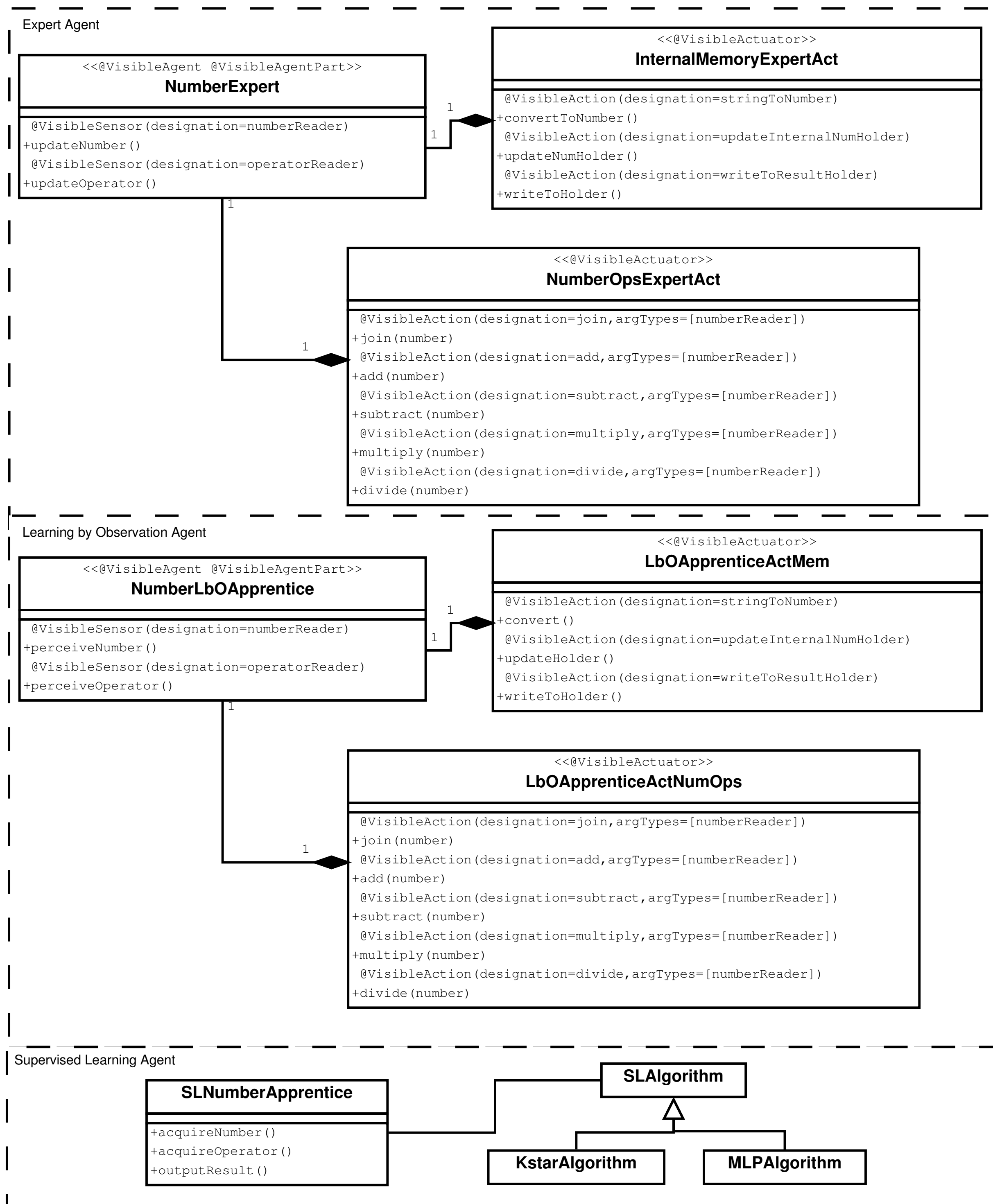


Figure D.5: The class diagram of the expert and the apprentice agent in the number calculator scenario

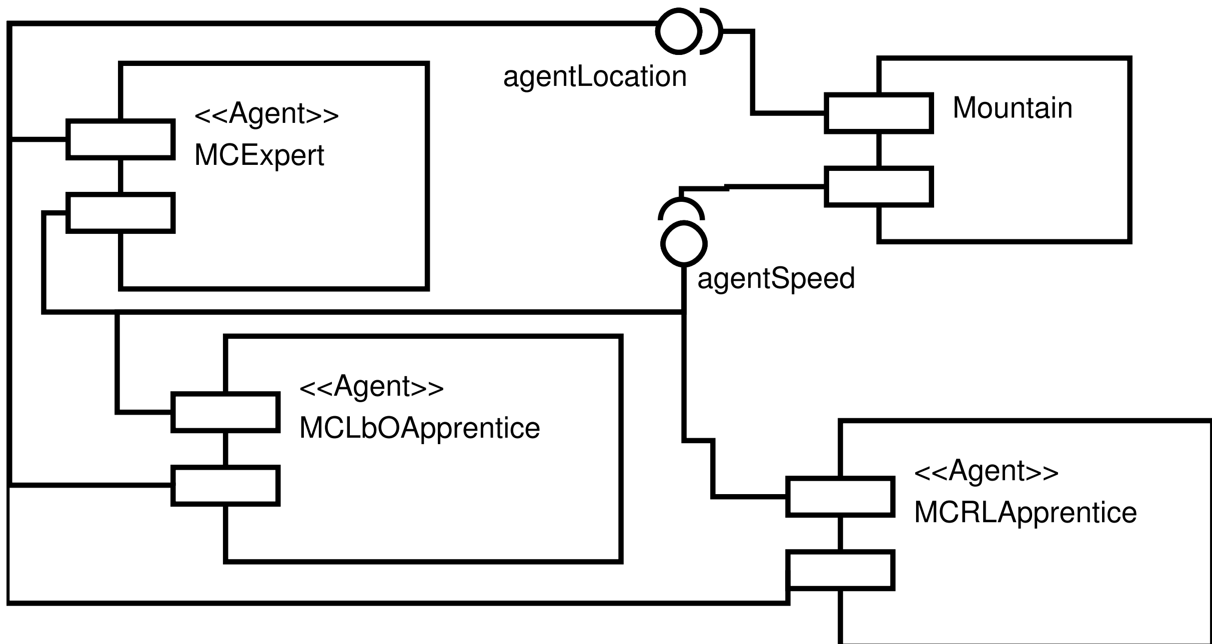


Figure D.6: The component diagram of the elements of the mountain car scenario

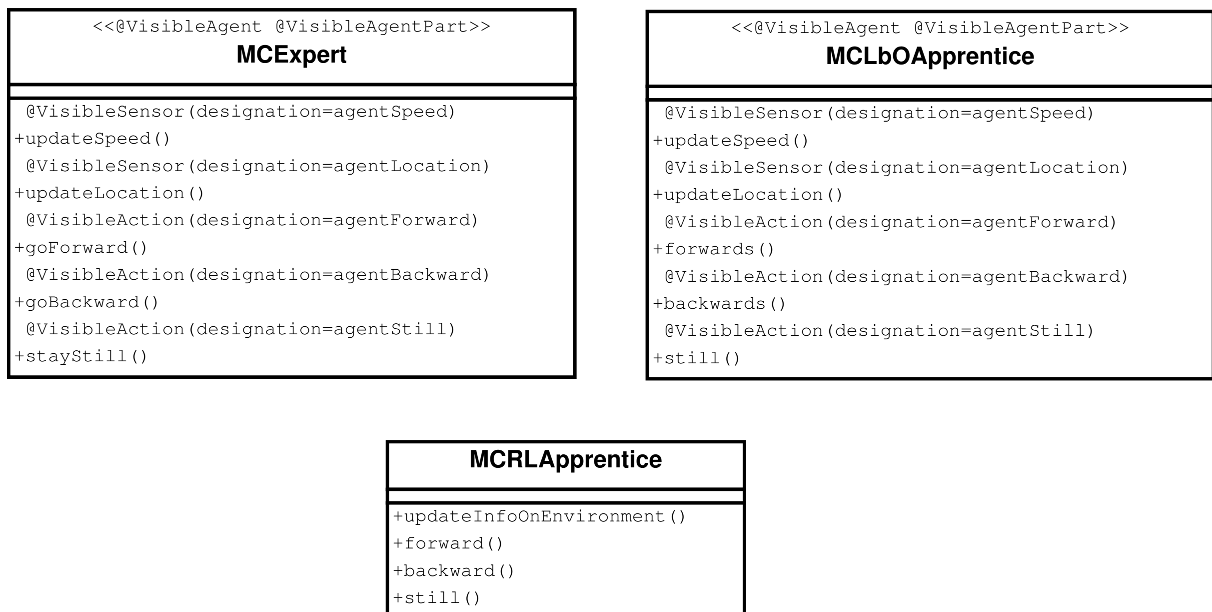


Figure D.7: The class diagram of the expert and the apprentice agents in the mountain car scenario