# Software Image for Learning by Observation

Paulo Costa[1] and Luis Botelho[1]

Instituto de Telecomunicações/ISCTE-Instituto Universitario de Lisboa
Lisbon, Portugal.
paulo.costa@iscte.pt; luis.botelho@iscte.pt

**Abstract.** As it is the case in human societies, software agents could also use learning by observation as an important method of knowledge transference between experts and apprentices even if they have different knowledge representations. However, observation requires that agents and the actions they perform be visible. In this paper, we propose the novel notion of software image that allows software agents, as well as their actions, to become visible to other agents. The software image was designed to accomplish two purposes: allow agents to locate similar experts to observe and provide training examples for the learning by observation algorithm.

**Keywords:** Software agents, expert agent, software image, learning by observation, learning architecture

## 1 Introduction

The ability of software agents of learning from each other opens a new perspective on agent development. In the software world it would seem that such effect would easily be achieved if the expert agent would directly convey knowledge to the apprentice. However this would only work if agents share a common knowledge representation.

The PhD research reported on this paper proposes a solution through which software agents may learn by observation, even without a common knowledge representation method. The proposed agent architecture was inspired in superior mammal learning by observation.

Learning by observation, one of the most powerful socialisation and knowledge acquisition mechanisms [14, 3], requires the ability to see. Unfortunately, software agents cannot see each other hence they cannot learn by observation. This paper presents an innovative agent architecture with software image that allows agents to see each other and hence learn by observation. However, the paper is focused only on the agent software image; not on the learning process.

In superior mammals, learning by observation requires an initial identification of similar entities to observe [3]. In this initial stage, the individual must recognize similar ones from which to learn. Similar entities share similar structures and capabilities and are able to learn by observing each other's behaviours. The proposed agent architecture with software image also provides agents with recognition capabilities.

After recognizing a similar expert, apprentices observe them while doing some task and learn how to perform it. In this case the software provides training examples that can be used by apprentices' learning by observation algorithms.

Machado [11] describes the software image as a mean to provide software agents with a visible representation of their bodies and actions. The concept of software image (SwI) used in this research extends her work with the capacity to store the agent's perception and past events in addition to its visual image.

Superior mammals use vision sensors to observe the expert's bodies and their actions and learn from those observations, but although advances in computer vision enable computers to have vision systems similar to humans, a simpler approach was used for the software image - meta-data. Apprentices use specialized sensors to extract meta-data stored on the expert agent software image and learn new abilities.

In addition to the described enhancements to the software image definition, we have built the software image toolkit (SIT), which allows creating agents with software image. The toolkit is domain independent and contains all the required tools to generate and update the agent's software image with minimal intervention from the agent developer.

The next section presents a survey on research on learning by observation. Section 3 presents a formal representation of the approach proposed in our research. Section 4 presents the conceptual view of the software image. On Sect. 5 we briefly describe the software image toolkit. Section 6 demonstrates the application of the software image. On Sect. 7 we show a performance diagnostic that lend support to our design choices for the software image matching algorithm. Section 8 presents the acknowledgements. Section 9 terminates with conclusions and future work.

## 2 Related Work

Learning is an essential characteristic of intelligent beings. A computer program is able to learn if its performance on a set of tasks improves with experience [12]. Learning algorithms can be organized in two types: supervised learning and unsupervised learning.

Supervised learning creates mappings between inputs and outputs, whose correct values are provided by a supervisor. Training sequences, provided by the supervisor, are used as basis for optimization. Unsupervised learning determines how a dataset is organized. These algorithms try to find regularities in the input data to extract knowledge from them [1].

Several authors [2, 4] define learning by observation as a subset of supervised learning, where policies are generated by observing, retaining and replicating the behaviour executed by an expert. The capacity to observe and imitate the movements of others is among the least common and most complex forms of learning [13].

Learning by observation can be explained under the human and superior animals social interaction mechanisms. Animals and humans take benefit from the

experiences of others by learning what they observe from them. Bandura [3] emphasizes this aspect on his social learning theory. According to Dautenhahn [10], the social intelligence hypothesis claims that some intellectual capacities evolve out of a social domain.

As Argall and her colleagues describe in their survey [2], implementations of learning by observation algorithms are usually related to robotics. Robot agents allow a more natural interaction with humans, making it easier to use human demonstrators as experts. Robot agents also have the advantage of being able to see each other, when provided with proper vision sensors. Software agents do not have this capacity, making it impossible for them to identify similar software agents from which to learn. Additionally, software agents can only observe the effects of the actions performed in the world. They cannot observe the action being executed by another agent.

Observing an expert software agent performing its actions has advantages when learning actions that change internal features in the agent, or whenever its effects are not visible in the environment (e.g. an agent becomes afraid each time it passes near a red object, learning a communication protocol), when the same effects could be achieved by different alternative actions but using one of them is clearly better than using others (e.g. the use of a set of sums instead of a simpler multiplication), and also in situations in which the agent must perform actions whose effects it doesnt know before hand.

Machado [11] proposes a solution to this problem. She designed a software agent architecture with software image, representing the parts of the agent's software body with visual appearance. Her proposal is based on Botelho and Figueiredo's work [5] on architectural principles for embodied agents.

Our approach builds on Machado's software image, extending it with the capacity to store the agent's perception in addition to its visual image. Our proposal also stores a limited history of the agent dynamic image in past interactions. This will allow the learning algorithm to rely on more data than if it were limited to the observed agent instantaneous image.

The agent's visual image was redesigned to include the agent's input mechanisms (the sensors) as visible elements. Sensors provide agents with the perception of their surroundings, which is an important factor for the decisions they make. The body part matching algorithm proposed by Machado was also extended to include sensor matching. Matching also becomes the first stage of the learning by observation algorithm.

## 3 Formalization and Knowledge Representation

Software agents participating in learning by observation can have one of the following roles: the expert agent and the apprentice agent. The expert agent detains the knowledge on how to achieve desired goals. The apprentice agent learns, through observation, the actions it needs to perform to achieve shared goals. Both expert and apprentice are visible software agents (VSA), that is, software agents with software image.

Visible software agents follow the conceptual design of an agent: a mechanism that collects information from the environment and generates a behaviour by analysing the collected information and reflecting on its internal state [9]. In our proposal for the software image concept, the agent's visible appearance follows the conceptual design of an embodied agent from Brooks [6].

Under Brooks vision, agents are regarded as blocks of problem solving components (1), called agent parts (AP). Agent parts may be seen the same way as living beings parts, with the particularity that each agent part has its own control mechanism. This kind of design follows Brooks [7] ideas on solving complex problems by breaking them in smaller and simpler problems, solved by each agent part, avoiding centralized control components.

$$\text{VSA} \equiv \{AP_1, \cdots, AP_n\} \ . \tag{1}$$

Six distinct types of elements make out the building blocks of agent parts (2). They represent the agent's input mechanisms (sensors $S \equiv \{S_1, \cdots, S_m\}$), output mechanisms (actuators $AC \equiv \{AC_1, \cdots, AC_n\}$) and a set of internal constituents such as visual attributes $VA \equiv \{VA_1, \cdots, VA_p\}$, internal agent parts $IP \equiv \{IP_1, \cdots, IP_q\}$ internal attributes $IA \equiv \{IA_1, \cdots, IA_r\}$and the control mechanism $Cm$.

$$AP \equiv \{S, AC, VA, IP, IA, Cm\} \ . \tag{2}$$

Four of those six types of elements are visible, meaning that they have a representation on the software image: sensors, actuators, visual attributes and internal agent parts. Sensors are atomic elements whose purpose is to feed the agent part with information from the environment. Actuators represent the array of atomic operations, or actions $a$, the part is able to perform ($AC_i \equiv \{a_1, \cdots, a_n\}$). Visual attributes are atomic elements that represent the types of visual messages that an agent is able to provide (e.g. a colour, an emotion). Internal agent parts represent functionality blocks inside the agent part. They are made out of the same elements as agent parts, with the peculiarity of being under an agent part instead of under the agent.

In each problem addressed in this research, we consider $E$ the collection of world states, and $A$ ($a \in A$) the collection of all possible actions. Action selection (AS) is the agent control function. It selects the action the agent will perform in a given state, $AS : E \rightarrow A$. The action selection function is implemented by the agent part's condition-action rule engine.

Being $NS : (E \times A) \rightarrow E$ a function that maps a given state and the action that is performed in that state into a new state, the state at time t can be computed from the state at time $t-1$ by the function $Next : E \rightarrow E$, according to (3).

$$e_t = Next\left(e_{t-1}\right) = NS\left(e_t, AS\left(e_{t-1}\right)\right), \quad \text{where} \quad e \in E \ . \tag{3}$$

Agent parts perceive the environment through their sensors, providing them a limited view of the current environment state $Z$ ($Z \subseteq E$). Condition-action

rule engines choose the actions the agent parts should take (4). A rule is satisfied, meaning it is one of the rules that may trigger, if the conjunction of conditions derives from the union of the parts perception Z and its current internal state (IS), as described in (5).

$$Cond_1 \wedge Cond_2 \wedge \cdots \wedge Cond_m \Rightarrow a_1, \cdots, a_n \ . \quad (4)$$

$$(Z \cup IS) \vdash Condition_1 \wedge \cdots \wedge Condition_m \ . \quad (5)$$

Agent goals are represented by the condition $G$, which represents the set of states where the agent goal is accomplished. In a typical setting, experts and apprentices may share different goals and is up to the apprentice to observe several experts until one of them performs a set of actions that leads to the apprentices desired goal. To remove the complexity of finding a proper expert from this research, the original problem is simplified so that experts share the same goals as apprentices, or, in the worst case, the expert's goal requires a set of actions whose intermediate result represents the apprentice's goal.

## 4 The Software Image Concept

Learning by observation is only effective when the observed expert and the apprentice share visible structures and capabilities. The initial stage of identification of similar experts provides clues for solving correspondence problems that otherwise could affect the ability to learn [2].

The correspondence problem amounts to establishing the mappings between expert and apprentice structures, allowing a common representation of agent perception and actions. The software image provides agents with the ability to recognize each other and to create embodied mappings between expert and apprentice [2]. Thus, the software image serves two purposes: create and provide agent representations in the same way as body part representations exist in the human brain [14], and create and provide representations for the agent's perception and the actions it performs. Agent body part representation is provided by the static image. Agent perception and action representation is provided by the dynamic image.

The software image is an independent layer of the agent architecture. Our approach to create agents with software image relies on a set of automated mechanisms that create domain independent agent representations with minimal intervention from the agent developer.

As Fig 1 shows, the static image represents the agent's visible properties (agent parts, sensors, actuators, visual attributes and internal agent parts). It is a reflection of the agent's visible body.

As the UML class diagram in Fig. 1 shows, the visual representation of an agent, as described in the static image, is made out of agent parts. Each of these agent parts is on its turn, an aggregate of sensors, actuators, visual attributes and internal agent parts. Actuators are, on their turn, aggregates of actions.

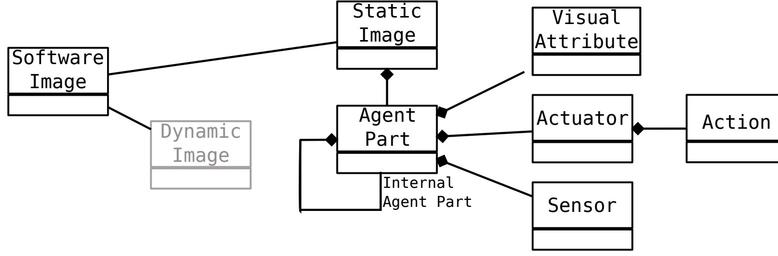


**Fig. 1.** General representation of the static image

Sensors, actions and visual attributes are atomic, meaning they have no internal constituents.

In order to make things simpler, atomic elements are represented according to a shared ontology, which facilitates the identification of similar agents, that is, agents with shared visible structures and capabilities.

Visual software agents may have other non-visual features, apart from the ones emphasized by the static image. The control mechanism of an agent part is an example of one of those features. In our approach for learning by observation, agents only need to share similar visible features to learn by observation. This allows learning between agents with different internal features.

Our learning approach uses the information provided by the expert agent's dynamic image as a set of training data for learning algorithms (state-action pairs). As Fig. 2 shows, the dynamic images is a set of snapshots that show what happened to the expert agent within a limited period of time. Snapshots are taken each time the agent makes a decision.

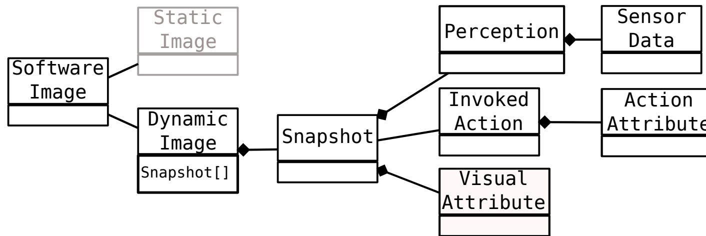


**Fig. 2.** General representation of the dynamic image

Each snapshot contains information of the agent's perception when the decision was made, the instance values of the agent's visual attributes and a representation of the action instances (as they are invoked in that particular situation) along with the set of attributes used for the invocations. This information is retrieved by a specialized mechanism in the software image and it is stored on the expert's software image, constantly updating it. These specialized mechanisms

include a sensor that captures the necessary data from the observed agent. The information stored on the dynamic image makes out the training data (state-action pairs) required for the learning by observation algorithm.

Unlike visualizations in the physical world, software visualizations can be stored with all the necessary information to allow playbacks with the same quality as the original observation. That is, the agent dynamic image comprises historical information (the stored set of snapshots) because historical information gives apprentice agents the ability to observe the expert's past actions, perceptions and visual attributes, allowing them to rewind the observation. This improves the apprentice's learning curve, since it is possible to review the training without having to wait for the expert to perform the same actions again. Historical information consumes a large amount of memory, thus the dynamic images historical records will only be able to store a limited amount of information.

## 5 The Software Image Toolkit

The software image toolkit (SIT) is a by-product of our research and was designed to facilitate the development of visible software agents. It provides agent developers with a set of tools to automatically create and update a domain independent agent software image. As explained in Sect. 4, the agent software image allows software agents both to identify expert agents with a similar visible structure and capabilities, from which to learn, and provides the training set of state-action pairs necessary for the learning algorithm. Using the dynamic image history, agents may also rewind what they have observed enabling them to improve the learnt action control rules, if necessary. As Fig. 3 shows, the SIT provides agent developers with four automated and domain independent mechanisms (software image building mechanism, a dynamic image update mechanism, a perception mechanism, a static image matching), and a set of interfaces for the software vision sensor.

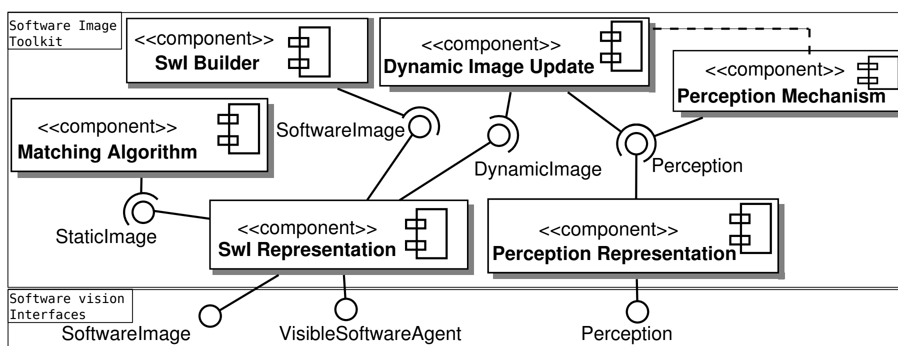


**Fig. 3.** Component view of the software image toolkit

Figure 3 also shows that the SIT provides representations for the software image and agent perception. Aspect oriented programming (AOP) and code introspection are the main technologies behind the SIT tools. They allow minimal interventions in the software agent code, reflected on the use of code annotations, a special kind of meta-coding, to identify the agent's main class, agent part implementation classes, sensor and action methods, visual attributes, sensor implementation classes and actuator implementation classes. All of these annotations are listened by the SIT mechanisms through aspects. Code introspection is used to gain access to the agent's code and build its static image. The building mechanism makes use of the code annotations to identify the agent's visual features.

The dynamic image update mechanism listens for calls on annotated action methods and updates the agent's dynamic image with a new snapshot each time the agent takes a decision. The SIT provides each agent with a perception array, where agent perceptions are stored. The relations between sensors and positions on the perception array are also stored, to guarantee that sensor updates are correctly replaced in the perception array. The perception mechanism listens for calls on annotated sensor methods, updating the agent's perception array, each time sensor information is acquired.

The static image matching algorithm compares the static images of two agents to find out their differences. It is responsible for determining if the apprentice and the expert agents share relevant structural elements and capabilities. Matching happens only in the static image, agent dynamic behaviour is not involved in this process.

The matching process was inspired in superior mammal recognition of familiar structures when observing similar entities. The algorithm is a recursive method, based on tree comparison algorithms. The agent's static image is regarded as the root of a tree whose branches represent the agent parts. In their turn, agent parts branch into actuators and internal parts. The tree leafs represent sensors, actions and visual attributes.

From the learning by observation perspective, apprentices can learn from experts as long as the apprentice is able to map its visible structures with the expert's visible structures. To do this mapping, each branch of the agent's static image is matched with the branches of the other agent, to find correspondent leafs in the matched branches. Optimizations in the matching algorithm prevents matching between different types of branches (e.g. a branch that represents an agent part can only be matched with branches that represent agent parts), and allow an immediate return of a no matching branch whenever an unmatched leaf is found.

The matching algorithm is also optimized to allow apprentices to have extra parts that do not exist on the expert. If all the experts parts can be matched with one of the apprentices parts, the apprentice is able to learn from that expert, even though the apprentice has other body parts that do not exist on the expert. This is illustrated in e.g. 1.

*Example 1.* Apprentice agent A is made up of parts $\alpha, \beta$ and $\omega$. Expert agent E is made up of parts $\mu$ and $\pi$. If the static image representation of $\alpha$ and $\mu$ is identical to the static image representation of $\beta$ and $\pi$, agent E is similar to agent A, from the perspective of agent A. This is true even though E does not have a part corresponding to $\omega$.

The software vision sensor is responsible for gathering information from the expert's software image through the interfaces provided by the SIT. In supervised learning algorithms, agents are presented with a set of labelled training data, usually state-action pairs, and have to learn an approximation to the function that produces the same results [2]. In the specific case of learning by observation, the set of training data is provided by the experts dynamic image.

## 6 Example of a Visible Software Agent

The following test scenario was developed to test the capabilities of software agents with software images and to test the SIT mechanisms. In this scenario the expert agent is presented with a set of even numbers, from two to eight, and has to decide whether to take the number or discard it.

The expert's goal is to achieve the number 24 out of the taken numbers, summing them up, and to print it. Numbers are presented one at a time to the expert agent. Each time a number is presented a call to the control mechanism is made, to choose the actions the agent must take.

The scenario's environment is made out of a random number generator and a print area. The expert agent, whose static image is represented in Fig. 4, was developed with the SIT. It is made out of a single part, responsible for recognizing the numbers, deciding if the number is taken or discarded, and presenting the number twenty four in the print area.

As Fig. 4 shows, the agent's single part (AP1) has one sensor that provides the agent with the ability to recognize numbers from the random number generator (*provider:even_number*). It has one internal attribute, the *storage* attribute, where the taken numbers are stored. The part also has two actuators: the first one (A1) provides the expert with two actions; *take* (*store(even_number) : self*) and *discard* (*reject(even_number)*); the second one (A2) provides the expert with the *print* action (*print(stored) : print_area*).

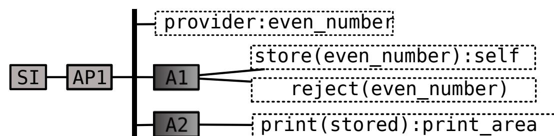


**Fig. 4.** Representation of the expert's static image

The *take* action picks up the number provided by the random generator and adds it to the agent's internal attribute. The *discard* action ignores the number provided by the random generator. The *print* action prints the number stored in the *storage* attribute in the environment's print area. The part's rule engine makes the decision to call these actions depending on a set of condition-action rules.

An observer agent was specially built to validate the generated expert agent's software image. The observer agent makes use of the software vision sensor to observe the expert agent. It was able to accomplish the following set of tasks:

The first two tasks required the observer agent to find the expert by matching its own image with the expert's software image. The first task was accomplished while the expert was running on the same computational multi-agent platform as the observer agent. The second task was accomplished while the expert was not running in the same computational platform as the observer.

The third task required the observer to identify a particular feature on the expert, its *take* action. The observer was able to read the expert's software image and find the required action.

The fourth task required the observer to acquire a training sequence from the expert's dynamic image. The task was accomplished and the observer was provided with a training sequence that allowed it to realize that when the sequence $\{6, 8, 4, 8, 2, 6, 4\}$ is shown, it may take the first three, the fifth and seventh numbers, and discard the rest. After this sequence of actions, the print action can safely be called, which will result in goal achievement.

Feeding the learning by observation algorithm with this single example would enable an apprentice agent to learn how to perform the same way as the expert agent, in this specific situation. According to Argall and her colleagues [2], apprentice performance is limited by the quality and quantity of information provided by the expert.

In this particular case the problem could be solved with subsequent observations. After a few observations, it is expected that the apprentice's response is able to cover a large set of possible sequences of numbers.

## 7 Matching Algorithm Performance Tests

Since matching is the initial stage of our learning approach, its performance is important since it affects the overall performance of learning. Tree comparison algorithm performance is affected by the number of branches and the depth of the compared trees. In the matching algorithm, performance is affected by the number of agent parts, number of visible elements (sensors, actuators, visual attributes and internal agent parts) and depth of internal agent parts.

Figure 5 shows the results of three tests performed to the matching algorithm. They determine how the number of parts, elements and depth affect the performance in terms of processor occupation time.

In the first test, we increased the number of agent parts (one at each step, until reaching twenty one parts) while maintaining the same number of elements

(five sensors, five actuators with a single action, five visual attributes and two internal parts) and the same levels of depth of internal parts (five levels). In the second test, we increased the number of each kind of element (one of each at each step until reaching twenty one sensors, actuators with a single action and visual attributes) maintaining the same number of internal parts (two on each agent part), the same levels of depth of internal parts (five levels) and agent parts (five agent parts). In the third test, we increased the levels of depth of internal parts (one level at each step until a depth of 21 is reached) while maintaining the same number of elements (as in the first test) and the same number of agent parts (five agent parts).

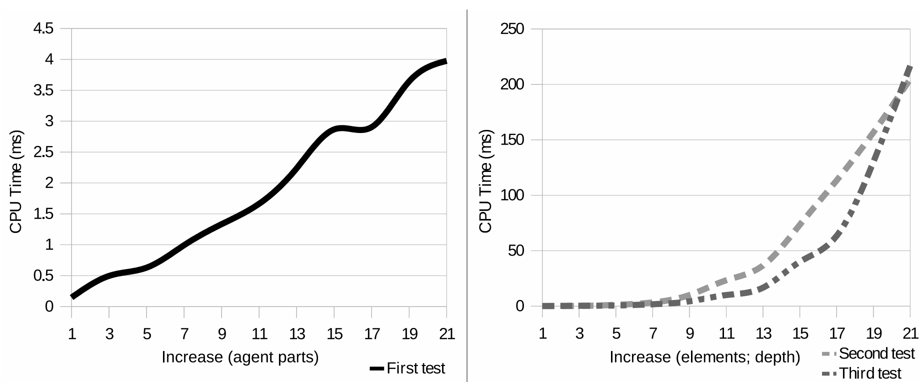


**Fig. 5.** Matching algorithm performance test results

In each test, we have computed the average values of multiple matching operations (1000 matches) between two identical software images. Comparing identical agents is the situation that involves the largest number of iterations, since it is necessary to match all elements with each other to prove both images are equal. For each matching operation, a new pair of equal software images was randomly generated, providing a large number of different possibilities. The random generation algorithm ensures minimal deviation of the non-varying parameters (depth, number of visible elements or number of agent parts).

Results are presented in two plots to allow a detailed observation of the data from the first test. The $x$ axis represents the tests progression (the increments of the number of parts, elements and depth). The $y$ axis represents the average CPU time, in milliseconds, taken between the beginning and the end of the matching process. As expected, Fig. 5 shows that increasing the three described factors causes an increase in processor time, but the effects of the increase in the number of elements and the levels of internal parts are the most significant. Another important aspect to take into consideration is the fact that, when the number of elements and the level of internal parts reach a certain value (increment of 13),

new increments of any of those two parameters lead to an exponential growth of the consumed processor time.

The matching algorithm diagnostic shows that building agents with a large number of small parts (with a small number of elements and low levels of internal part 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.

## 8 Acknowledgements

This paper was submitted to SDIA 2011 - 3rd Doctoral Symposium on Artificial Intelligence and reports PhD research work for the Doctoral Program on Information Science and Technology of ISCTE-IUL (University Institute of Lisbon). The research is supervised by Professor Luis Botelho, started on September 2009 and is planned to finish on August 2012. 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.

## 9 Conclusions and Future Work

The software image presented in this paper is the result of the first stage of the PhD research on learning by observation in software agents. The definition of a learning algorithm for the agent architecture represents the second and final stage of the research.

Currently, the research has produced a prototype agent architecture for learning by observation. It integrates the software image with a learning algorithm. It was also possible to perform tests on learning algorithm approaches. Their results narrowed the algorithm choice to classification and planning approaches for learning by observation [2]. A set of by-products was also created throughout the research, such as the SIT, several test scenarios as the one presented in Sect. 6 and the learning algorithm tests.

Future work consists on the completion of the second stage of the research. The main decision left to make is the choice of the type of algorithm to use. A CBR algorithm is one possible alternative. Another alternative is building a dynamic plan that is updated each time the apprentice observes a new behaviour on experts. The plan can be turned out into rules and applied to the apprentice's control mechanism. It can also be used directly by obtaining the correct path from the closest match to the apprentice's perception.

## References

1. Alpaydin, E.: Introduction to machine learning. MIT Press, Cambridge Mass. (2004)

2. Argall, B.D., Chernova, S., Veloso, M., Browning, B.: A Survey of Robot Learning from Demonstration, vol. Robotics and Autonomous Systems, pp. 469–483. Elsevier (2009)
3. Bandura, A.: Social learning theory. Prentice Hall (1977)
4. Billard, A., Dautenhahn, K.: Experiments in learning by imitation - grounding and use of communication in robotic agents (1999), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7685
5. Botelho, L.M., Figueiredo, P.: What your body and your living room tell my agent (2004), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.81.3795
6. Brooks, R.A.: Elephants don't play chess. Robotics and Autonomous Systems 6, 3–15 (1990)
7. Brooks, R.A.: Intelligence without reason. pp. 569–595. Morgan Kaufmann (1991)
8. Calinon, S.: Incremental learning of gestures by imitation in a humanoid robot. In: In Proceedings of the 2007 ACM/IEEE International Conference on Human-Robot Interaction. pp. 255–262 (2007)
9. Costa, E., Simoes, A.: Inteligencia Artificial. FCA (2008)
10. Dautenhahn, K.: Trying to imitate - a step towards releasing robots from social isolation. Proceedings of From perception to action conference pp. 290–301 (1994)
11. Machado, J.a.: Imagem Visual do Corpo de Software: Aquisição de Vocabulário por Observação [Software Body Visual Image: Acquiring Vocabulary by Observation]. Masters degree thesis, ISCTE (2006)
12. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
13. Moore, B.: Avian movement imitation and a new form of mimicry: tracing the evolution of a complex form of learning. Behaviour 122, 231–263 (1992)
14. Ramachandran, V.S.: The emerging mind: the Reith Lectures 2003. Profile Books (2003)