# Building Learning by Observation Agents Using jLOAF

Michael W. Floyd and Babak Esfandiari

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, Ontario, Canada

**Abstract.** The environments an agent is situated in or the behaviours it is required to perform may change over time. Ideally, an agent should be able to move to a new domain without requiring significant changes from the agent's designer. We describe our framework jLOAF (Java Learning by ObservAtion Framework) that attempts to minimize the amount of effort required to move a learning by observation agent into a new domain. We will demonstrate how an agent can be created using jLOAF and show how it is able to learn to perform a new behaviour.

## 1 Introduction

Software agents and robots are often situated in complex environments that are partially observable, and they need to make their decisions in real-time. The tasks these agents are required to perform can change over time, requiring them to regularly learn new behaviours or be retrained. *Learning by observation* is an alternative approach to traditional agent programming that transfers the burden of training from the programmer to the agent. Instead of being explicitly trained by the programmer, the agent learns by watching an expert perform a desired behaviour. The agent observes how the expert reacts, in the form of actions, to sensory inputs and then trains itself using the observed data.

A general-purpose learning agent should be able to be deployed in a variety of environments with a wide range of behaviours and goals. Ideally, such an agent should be able to learn their behaviours without being explicitly told the task they are learning or their goals. Our approach to developing learning by observation agents, jLOAF (Java Learning by ObservAtion Framework), was designed in such a way as to attempt to avoid hard-coding any domain knowledge that may specialize the agents to any specific task. This paper will serve to highlight our design philosophy while providing a demonstration of how a learning by observation agent can be developed with jLOAF.

## 2 Learning by Observation Agent Design

In a learning by observation system, the learning agent observes and records the interactions between an expert and the environment (Figure 1). In a case-based reasoning system, this involves recording the sensory inputs $S$ that the

expert receives (*the problem*) and the resulting actions $A$ performed by the expert (*the solution*) as a case $C$. This process of passive observation is performed in the majority of case-based learning by observation systems [1, 3–5] as well non-CBR approaches [2, 6] (although these systems store the observations as training examples and not as cases).

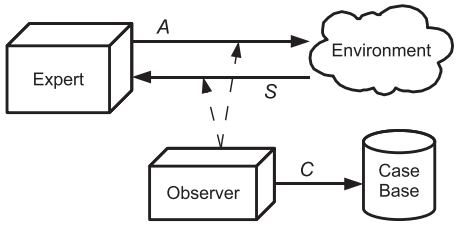


**Fig. 1.** Observation of an expert interacting with the environment

One of the most significant bottlenecks in deploying an agent in a new domain is allowing the agent to interact with the environment of that domain. This includes modifying the agent so it can properly sense inputs and perform actions. In our approach, we clearly separate these environment interfaces from the central reasoning module of the agent (Figure 2). The *Perception* module converts raw sensory data into a form that is understandable by the *Reasoning* module (an input problem). The Reasoning module then uses the input problem, from the Perception module, to query the case base and determine an action to perform. This action is then used by the *Motor Control* module to actually perform the action.

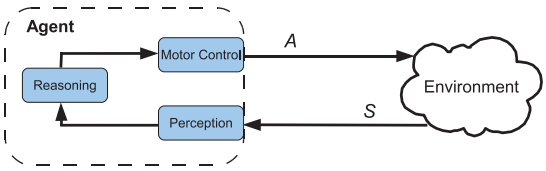


**Fig. 2.** The interfaces that allow an agent to sense and act on its environment

### 2.1 Reasoning

Our goal is to have a Reasoning module that can be utilized, unchanged, in a wide variety of domains. For example, an agent that learns soccer will have a specific Perception and Motor Control module that allows it to play the game. However its Reasoning module is the same reasoning module that it would use

if it was trying to learn another behaviour (like controlling a robot). The Reasoning module will only change the case base it uses, not any of the underlying algorithms.

In order to allow for such a reusable Reasoning module, all algorithms have been implemented to use a general model for sensory inputs (Figure 3) and actions (Figure 4). As long as the Perception module provides input problems in this format (and the Motor Control module accepts actions in this format) the Reasoning system is able to perform the necessary case-based reasoning.

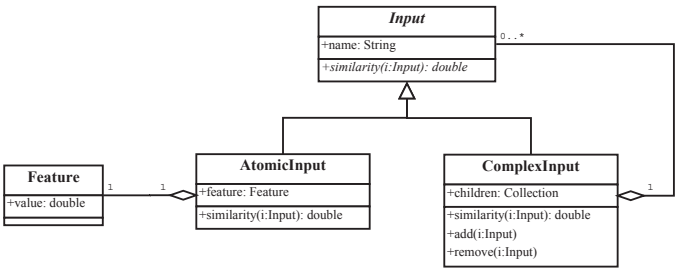


**Fig. 3.** Model of sensory inputs

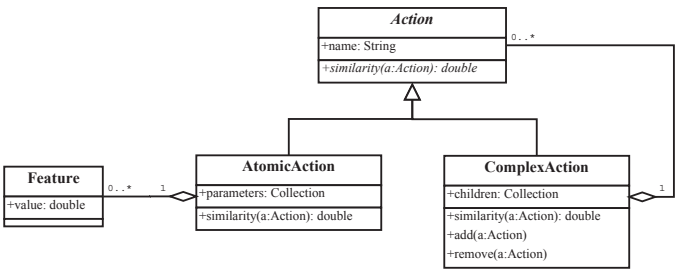


**Fig. 4.** Model of actions

## 2.2 Perception

The Perception module converts the raw sensory data into a form that is understood by the reasoning module (based on the sensory input model shown in Figure 3). The following sample code shows how this would be done when the raw sensory inputs are in the form of readings from two robotic sensors (a sonar and touch sensor):

```
public Input perception(double sonar, double touch){
    ComplexInput i = new ComplexInput("robotInput");
    AtomicInput s = new AtomicInput("sonar", sonar);
    AtomicInput t = new AtomicInput("touch",touch);

    i.add(s);
    i.add(t);
    return i;
}
```

The Perception module can also define what type of similarity metric the Reasoning module should use when comparing input:

```
    i.setSimilarityMetric(new Mean());
    s.setSimilarityMetric(new NormalizedDifference());
    t.setSimilarityMetric(new NormalizedDifference());
```

In this example, when comparing the input problem (the complex input $i$) to the problem portion of a case ($i2$) the Reasoning module will just need to call the similarity function:

```
    double sim = i.similarity(i2);
```

Since $i$ was set to use the Mean similarity metric, calling the similarity method will use that particular similarity metric (which will in turn calculate the mean similarity of the child inputs using their similarity metric *Normalized-Difference*).

### 2.3 Motor Control

The Motor Control module performs a mapping between the actions that are understood by the Reasoning module (the *Action* class) and the actual actions. In a simulated domain this would involve converting the action into a message that would be sent to the simulation server and in a physical domain it would involve causing the robot to perform the action.

The following code example shows a simple Motor Control module that sends action commands to a simulation server:

```
public void motorControl(Action a){
    if(a.name == "UP")
        send("move_up");
    }else{
        send("move_down"};
    }
}
```

During observation, the Perception and Motor Control modules are used to create the case base. Whereas the Perception module can be used as is, by converting the raw sensory data to input problems, the Motor Control needs to perform a reverse mapping from server commands to Actions. This requires defining a second method that performs the reverse mapping:

```
public Action observeActions(String s){
    if(s == "move_up")
        return new AtomicAction("UP");
    }else{
        return new AtomicAction("DOWN");
    }
}
```

## 3 Demonstration

Our demonstration will show how learning by observation agents can be built using jLOAF. This will include developing an agent to operate in a new domain, observing an expert in that domain and performing the learnt behaviour. We will also show that even though our framework was designed to avoid adding domain-specific information, such domain information can still easily be added if required. In addition to an in depth demonstration in a single domain (Tetris) we will also briefly show other domains (simulated soccer, space combat and physical robotics) that jLOAF has been deployed in[1].

## References


1. Gillespie, K., Karneeb, J., Lee-Urban, S., Muñoz-Avila, H.: Imitating inscrutable enemies: Learning from stochastic policy observation, retrieval and reuse. In: 18th International Conference on Case-Based Reasoning. pp. 126–140 (2010)
2. Grollman, D.H., Jenkins, O.C.: Learning robot soccer skills from demonstration. In: IEEE International Conference on Development and Learning. pp. 276–281 (2007)
3. Ontañón, S., Mishra, K., Sugandh, N., Ram, A.: Case-based planning and execution for real-time strategy games. In: 7th International Conference on Case-Based Reasoning. pp. 164–178 (2007)
4. Romdhane, H., Lamontagne, L.: Forgetting reinforced cases. In: 9th European Conference on Case-Based Reasoning. pp. 474–486 (2008)
5. Rubin, J., Watson, I.: Similarity-based retrieval and solution re-use policies in the game of Texas Hold'em. In: 18th International Conference on Case-Based Reasoning. pp. 465–479 (2010)
6. Thurau, C., Bauckhage, C.: Combining self organizing maps and multilayer perceptrons to learn bot-behavior for a commercial game. In: Proceedings of the GAME-ON Conference (2003)


[1] The video "Case-Based Imitation: A Sequel" from the 2010 AAAI Artificial Intelligence Video Competition shows demonstrations of agents created using this framework: `http://www.videolectures.net/aaai2010_floyd_cbi/`