A Case-Based Reasoning Framework for Developing Agents Using Learning by Observation

Michael W. Floyd and Babak Esfandiari
Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, Ontario, Canada

***Abstract*—Most realistic environments are complex, partially observable and impose real-time constraints on agents operating within them. This paper describes a framework that allows agents to learn by observation in such environments. When learning by observation, agents observe an expert performing a task and learn to perform the same task based on those observations. Our framework aims to allow agents to learn in a variety of domains (physical or virtual) regardless of the behaviour or goals of the observed expert. To achieve this we ensure that there is a clear separation between the central reasoning system and any domain-specific information. We present case studies in the domains of obstacle avoidance, robotic arm control, simulated soccer and Tetris.**

***Keywords*-learning by observation; case-based reasoning; games;**

I. 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. In order to allow the development of such agents, we propose a general purpose framework, jLOAF (Java Learning by ObservAtion Framework), that allows agents to *learn by observation* in real-world environments. 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 by associating the actions to the inputs.

Our framework aims to allow the development of agents in a variety of environments with a wide range of behaviours and goals. These agents should be able to learn their behaviours without being explicitly told the task they are learning or their goals. In order to achieve this, our framework was designed in such a way as to avoid hard-coding any domain knowledge that may specialize the agents to any specific task. Additionally, design decisions were made to make the framework usable by agents that have minimal domain knowledge, limited computational resources, real-time constraints and environments that are only partially observable. However, a limitation of this design is that more time must be spent on preprocessing in order to extract any necessary knowledge from observations or to ensure that agent can process the observations in real-time.

In the remainder of this paper we will describe our learning by observation framework and describe how it has been deployed in several different domains. Section II describes the observation process and how sensory inputs and actions are modelled. The internal case-based reasoning cycle of the framework is presented in Section III. Section IV examines how the case base can be preprocessed in order to help meet real-time constraints and perform learning. Case studies in an obstacle avoidance robot, a robotic arm, simulated soccer and Tetris are detailed in Section V. Areas of related work are discussed in Section VI followed by concluding remarks in Section VII.

II. Observation

An *expert* will interact with its *environment* by performing an action A after receiving a sensory stimulus S (Figure 1). Over a period of time a number of such interactions will occur, resulting in a *run* R of sensory stimuli and actions [1].

$$R : S_0 \overset{A_0}{\rightarrow} S_1 \overset{A_1}{\rightarrow} S_2 \overset{A_2}{\rightarrow} \ldots S_{t-1} \xrightarrow{A_{t-1}} S_t$$

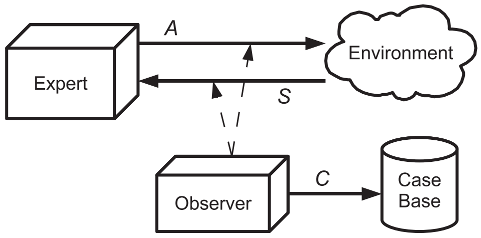

Figure 1. Observation of an expert interacting with the environment

For experts that are not *situated*, if their behaviour is independent of the environment, their actions are constant and are not influenced by any sensory stimuli:

1082-3409/11 $26.00 © 2011 IEEE
DOI 10.1109/ICTAI.2011.86

$$A_t = constant$$

However, most agents are situated and reason using sensory stimuli from the environment. If the expert is *reactive*, it will select an action to perform based only on the most recent stimulus received from the environment:

$$A_t = f(S_t)$$

However, if the expert's behaviour also depends on an *internal state*, then the task-relevant part of the state can be equivalently expressed and retrieved via the set of all distinct runs [1]. For state-based experts, the entire run may be used when selecting an action to perform:

$$A_t = f(S_t, A_{t-1}, S_{t-1}, A_{t-2}, S_{t-2}, \dots)$$

An agent that learns by observation acts as an *observer* and watches the interactions between the expert and the environment (Figure 1). The observer records the stimuli that the expert receives from the environment and the resulting actions that the expert performs. These observations can potentially be used to update a training model or, since our framework makes use of case-based reasoning [2], recorded as a case C and stored in the case base.

A learning agent will likely observe the expert over a period of time in which many interactions occur. This allows for a case C_t, observed at time t, to be created not only with the current stimulus S_t and current action A_t, but instead with the current action along with the entire run R_t that led up to it:

$$C_t =< R_t, A_t >$$

Such a case representation ensures that there is sufficient information to represent the reasoning process of both reactive and state-based experts. It should be noted that each case need not contain the entire run of the expert. Instead, the temporal relation between cases can be exploited to reduce the amount of information that needs to be stored in each case. We redefine a case recursively as a triple containing the currently observed environmental stimulus, the observed action and the previously observed case C_{t-1}:

$$C_t =< S_t, C_{t-1}, A_t >$$

This case definition makes it possible to backtrack through past cases and reconstruct a fragment of the expert's run or even the entire run. Using the link to case C_{t-1}, S_{t-1} and A_{t-1} can be accessed along with the link to case C_{t-2}.

The sensory stimuli received by the expert, and recorded in cases, can come in many forms. If the expert is a robot with simple sensors, the inputs would likely be in the form of numeric readings from the sensors. An expert who plays a board game would likely receive inputs regarding the current configuration of the board whereas an expert with object recognition capabilities might get inputs in the form of a set of visible objects. In order to account for the variability in what constitutes an expert's environmental inputs, the inputs come in two forms: *atomic inputs* and *complex inputs*. The modelling of inputs makes use of the *composite design pattern* [3] (Figure 2). An atomic input is used to model simple *feature* values. Each complex input is composed of a collection of other inputs, either atomic inputs or other complex inputs.

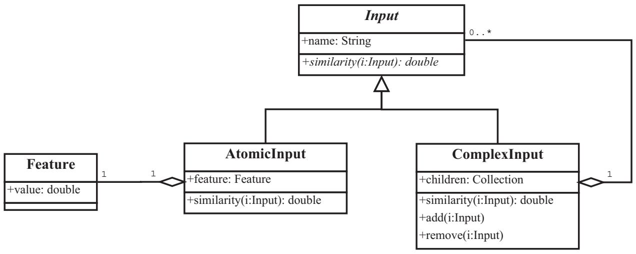

Figure 2. Model of sensory inputs

We take a similar approach when modelling *actions* (Figure 3). Actions can be atomic if they represent a single action or complex if they represent a sequence of actions. Each atomic action contains a collection of action features that represent the parameters of the action. This allows the modelling of very specific actions that have no parameters, like moving forward, or more general parameterized actions, like moving with a given direction and velocity.

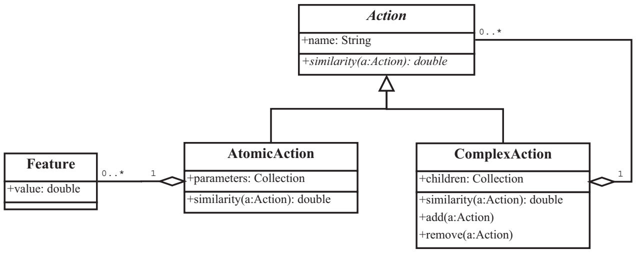

Figure 3. Model of actions

It should be noted that nowhere in our models is there any information about what the meaning of each input or action is. For example, no background information is provided to tell the observer that the touch sensor of a robot indicates the robot has come in contact with an obstacle or that a *forward* action is suppose to move the robot. More importantly, nowhere in these models, or anywhere in our framework, is there a need to explicitly represent the goals of the expert or add non-observable features related to those goals. This allows the same input and action models to have task-dependent meaning and to be reused for different experts even if those experts have different behaviours or work toward different goals.

III. Case-Based Reasoning Cycle

Agents that are developed using this framework use case-based reasoning (CBR) as their approach to learn by observation. CBR has been a popular approach for learning by observation systems [4], [5], [6], [7], [8], [9]. The primary benefit of using CBR is that the observations are not generalized in any way but are instead stored as concrete problem-solution pairs (cases). This is useful when dealing with state-based experts since it allows the entire run of the expert to be stored in the case base. Additionally, the solutions stored in cases can be complex in nature.

The *case-based reasoning cycle* [2] has four primary stages: retrieval, reuse, revision and retention. During the *retrieval* stage of the case-based reasoning cycle, input problems will need to be compared to cases in the case base in order to retrieve similar cases. Case similarity is calculated by combining the similarity of individual inputs. However, as we discussed in the previous section, different domains can potentially have very different sensory inputs and will therefore require different approaches to calculate similarity. Instead of encoding the specific similarity metrics in the retrieval algorithms, we make use of the *strategy design pattern* [3] (Figure 4). Each type of input can have its own method of similarity calculation, but the similarity calculation strategy is *decoupled* from the input to allow different strategies to be used, or for different features to (re)use the same strategy. This allows the various retrieval algorithms to be developed independently of the input model. The retrieval algorithm can call the similarity method of the input which will then delegate the calculation to the associated similarity metric.

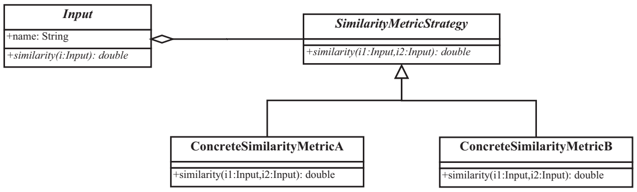
Input
+name: String
+similarity(i:Input): double
SimilarityMetricStrategy
+similarity(i1:Input,i2:Input): double
ConcreteSimilarityMetricA
+similarity(i1:Input,i2:Input): double
ConcreteSimilarityMetricB
+similarity(i1:Input,i2:Input): double

Figure 4. Each input has an associated similarity metric that will be used when its similarity method is called

Actions, like inputs, have associated similarity metrics. This may seem unintuitive since actions constitute the *solution* portion of the cases (what the reasoning system attempts to predict). However, it may be necessary to compare actions if they can be part of the case *problem*, like when the expert's entire run is compared during retrieval, or when comparing actions during solution *adaptation* (during the reuse part of the CBR cycle).

Since each complex input has a collection of sub-inputs, rather than a fixed number, our framework allows the representation of cases that have multi-valued features. This is important because most real-world agents operate in partially observable environments with noisy sensors. For example, consider a soccer player. Depending on where they look they will see a different number of opponents. At different points in time the number of visible opponents and their locations will change. If the player can not tell the individual opponents apart, it many not be possible to tell which opponent has moved to which location. Inputs that contain multi-valued sub-inputs will therefore require a similarity metric that is suited for multi-valued features and that is able to compare sets of values as opposed to single values [10].

Like retrieval, the *reuse* phase of the case-based reasoning cycle in our framework was designed to contain no domain information. This limits the reuse algorithms to those that directly copy solutions from retrieved cases or perform knowledge-poor adaptation. Any adaptation rules would need to be learnt from the observed cases.

The remaining two parts of the case-based reasoning cycle, *revision* and *retention*, have mostly been excluded from our framework. Both of these processes require some knowledge about the task being performed. During revision, a case-based reasoning system might evaluate the proposed solution or repair the solution. Without any knowledge of the task that the expert has demonstrated or direct feedback from the expert it would not be possible to know the quality of the proposed solution. Similarly, since the quality of the solution can not be measured, the system would not know which problem-solving episodes should be retained as new cases. However, during these stages the agent can make use of active [11] or mixed-initiative [12] learning. In these situations, the expert interacts with the agent to assist solving difficult problems or to correct the behaviour of the agent. This allows for the agent to learn over a period of time rather than only during a single observation session.

IV. Preprocessing

An agent will create a *raw case base* that will contain all the observed cases. This raw case base may include redundant cases, unnecessary features, or be lacking cases from certain regions of the problem space. Also, the case base might contain valuable information that could be mined and used to optimize performance. To account for this, the case base can be preprocessing before it is used in a deployed system.

Another motivation for preprocessing is that most agents, when deployed, will have limited computational resources and data storage. These computational limitations put a bound on the number of cases and features that can be used by the system. The case base can be preprocessed to guarantee the CBR cycle can be performed within any real-time limits and that the case base is small enough to fit in the agent's available storage.

Our framework allows for optional preprocessing steps, which are connected using a *pipe and filter* approach (Figure

5). Each preprocessing step takes in an *initial case base* and outputs a *processed case base* along with, optionally, some *extracted information*. The processed case base can then be used as input to another preprocessing step or used during execution by the system. Similarly, the extracted information can be used as settings for other preprocessing steps, settings for the deployed CBR system or simply recorded.

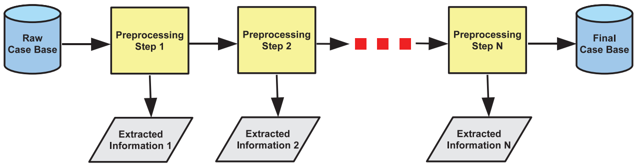

Figure 5. Preprocessing steps are connected using a pipe and filter approach

Several examples of preprocessing steps (and are provided in our framework) include:

- **Feature selection**: Identifying which features are important in order to optimize retrieval. Information could be extracted that relates to an optimal feature weighting or features could be removed from cases if they were found to have no importance [6].
- **Redundancy removal**: The size of the case base may be limited, due to computational or storage constraints, so it might be advantageous to replace clusters of identical, or highly similar, cases with a single representative case [6].
- **Case base analysis**: The analysis could find areas of the problem space that are underrepresented in the case base [11], [12]. This information can be used during future observation sessions to limit what new cases are recorded. This is an example of a preprocessing step that leaves the case base unchanged but records extracted information.
- **Case base restructuring**: This could involve converting a flat case base into a hierarchically structured case base [13] in order to reduce the time it takes to perform retrieval.

One final task that may be performed before the agent is deployed is evaluation. In our framework, evaluation is performed by giving the agent a series of testing problems to solve. The resulting actions of the agent can be compared to the known actions of the test problems to calculate performance metrics (like accuracy, precision, recall or f-measure) or the response time of the agent can be measured to see if it meets its real-time constraints (see [6] for an evaluation of several preprocessing algorithms for learning by observation agents).

V. Case Studies

Case studies in four domains will be used to demonstrate how our framework can be utilized. We will demonstrate how agents can be created in each of these domains and examine their performance when learning by observation.

A. Sensor-Based Agents

The first two domains we will examine involve controlling physical robots. The first is an obstacle avoidance robot that has a touch and sonar sensor. The robot moves forward until it detects an obstacle in front of it, using the sonar sensor, or determines it has come into contact with something, using the touch sensor. In situations where an obstacle is detected, it turns either left or right (it toggles its turn direction after every turn). If it comes into contact with something it moves backward. We will model the sensory input S_{AVOID} as follows:

$$\begin{aligned} S_{AVOID} &= < S_{touch}, S_{sonar} > \\ S_{touch} &= < f_{touch} > \\ S_{sonar} &= < f_{sonar} > \end{aligned}$$

The robot has a set $\mathcal{A}_{AVOID}$ of four possible atomic actions it can perform: *move forward* (A_F), *move backward* (A_B), *move left* (A_L), and *move right* (A_R).

$$\mathcal{A}_{AVOID} = \{A_F, A_B, A_L, A_R\}$$

The following sample code shows how this sensory model for this robot can be implemented with our framework:

```
 1  Input avoid = new ComplexInput("avoid");
 2  Input touch = new AtomicInput("touch");
 3  Input sonar = new AtomicInput("sonar");
 4  SimilarityMetricStrategy s1;
 5  s1 = new Mean();
 6  SimilarityMetricStrategy s2;
 7  s2 = new NormalizedDifference();
 8  avoid.setStrategy(s1);
 9  touch.setStrategy(s2);
10  sonar.setStrategy(s2);
11  avoid.add(touch);
12  avoid.add(sonar);
```

Initially, each of the sensory inputs are created (lines 1-3). Two similarity metrics are then created: *Mean* and *NormalizedDifference* (lines 4-7). As the names imply, the Mean similarity metric calculates the mean similarity of all child inputs and the NormalizedDifference similarity metric calculates the normalized difference between inputs. The Mean strategy is used by the complex input (line 8) and the NormalizedDifference strategy is used by both atomic inputs (lines 9 and 10). The complex input then adds both atomic elements as children (lines 11 and 12). It should be noted that in this example, for illustrative purposes, the similarity metrics and relationships between inputs need to be set each time the inputs are created. Instead, subclasses of ComplexInput and AtomicInput could be created for each input type in order to encapsulate these settings.

The second robot we examine is a robotic arm. This robot has three sensors: a colour sensor, touch sensor and sound sensor. It can perform five atomic actions: *move the arm forward* (A_{armF}), *move the arm backward* (A_{armB}), *stop the arm* (A_{armS}), *close the claw* (A_{clawC}), and *stop the claw* (A_{clawS}). Upon detecting a significantly loud sound on the sound sensor, the arm begins moving forward until the touch sensor signals it has come in contact with an object. If the colour sensor ever determines a red object is within the claw's grasp, the claw will be closed around the object and the arm will move in reverse. However, if it ever determines a blue object is within the claws grasp, it will not close but instead move the arm in reverse. We model the sensory inputs S_{ARM} and action set $\mathcal{A}_{ARM}$ of the robotic arm similarly to those of the obstacle avoidance robot:

$$\begin{aligned}
S_{ARM} &= < S_{colour}, S_{touch}, S_{sound} > \\
S_{colour} &= < f_{colour} > \\
S_{sound} &= < f_{sound} > \\
\mathcal{A}_{ARM} &= \{A_{armF}, A_{armB}, A_{armS}, A_{clawC}, A_{clawS}\}
\end{aligned}$$

There are two things that should be noted from our modelling of the sensory inputs and actions of these two robots. First, we see that although the robots were quite different there was still an opportunity to reuse a small portion of the model related to the touch sensor both robots had. Robots that make use of the same, or highly similar, sensors can therefore be modelled in similar ways. Both models could use the same similarity strategy, for their complex inputs, that calculates the mean similarity of the atomic inputs. Secondly, although we described the behaviour of the obstacle avoidance robot and robotic arm, the models we created are applicable to any robots that have the same sensors and actuators. If the obstacle avoidance robot was reprogrammed to follow objects it could still be observed and learnt from using the same model.

A software agent, using our framework, observed each of these robots and generated 500 cases for use as a case base. Additionally, each robot had 1000 extra cases generated for testing purposes. During the deployment, the learning agent used the case base to try and replicate the behaviour of the robots. Each of the testing cases, which had a known action, were given as input to the learning agent. By comparing the action selected by the learning agent to the known action of the test cases the accuracy was measured.

For the robotic arm, the learning agent achieved 100% accuracy. For the obstacle avoidance robot, the learning agent achieved 100% accuracy for forward and backward actions, but a lower accuracy (approximately 50%) for the left and right actions (an overall accuracy of approximately 75%). This occurred because the obstacle avoidance robot would toggle its turn direction, so the turn direction was related to its internal state and not any external information that could be observed. However, if the case retrieval takes into account the run of the agent instead of just the most recent sensory stimulus the accuracy of the left and right actions can be increased to approximately 73% (overall accuracy of approximately 87%).

B. Object Inputs

In this section we turn our attention to agents that have more complex sensory capabilities that allow them to observe the environment at a higher level of abstraction. Instead of receiving inputs in the form of sensor values, these agents are able to sense objects and their locations. In simulated soccer, like the RoboCup Simulation League [14], an agent's sensory input S_{SOCCER} contains the visible *balls* (S_{ball}), *teammates* (S_{team}), *opponents* (S_{opp}), *goal nets* (S_{net}), *boundary lines* (S_{line}), and *flags* (S_{flag}).

$$\begin{aligned}
S_{SOCCER} &= < S_{ball}, S_{team}, S_{opp}, S_{net}, S_{line}, S_{flag} > \\
S_{ball} &= \{S^1_{object}, \ldots, S^a_{object}\} \\
S_{team} &= \{S^1_{object}, \ldots, S^b_{object}\} \\
S_{opp} &= \{S^1_{object}, \ldots, S^c_{object}\} \\
S_{net} &= \{S^1_{object}, \ldots, S^d_{object}\} \\
S_{line} &= \{S^1_{object}, \ldots, S^e_{object}\} \\
S_{flag} &= \{S^1_{object}, \ldots, S^f_{object}\} \\
S_{object} &= < S_{distance}, S_{direction} >
\end{aligned}$$

Since objects can move in or out of the player's field of vision, the number of objects of each type can change over time. Therefore each object type is viewed not as a single-valued feature but instead as a multi-valued feature. This requires the use of a similarity strategy that can handle multi-valued features, using bipartite set-matching algorithms such as the one described in [10]. Depending on the expert that is observed, different types of objects may have different levels of importance. The following sample code shows how a case base can be mined to determine which types of objects are important and which can be removed from the cases:

```
1  CaseBase cb1 = CaseBase.load("soccer");
2  Preprocess p1 = new BinaryFS();
3  CaseBase cb2 = p1.preprocess(cb1);
4  Weights w = p1.getOptimumWeights();
5  Preprocess p2 = new FeatureFilter(w, 0);
6  CaseBase cb3 = p2.preprocess(cb2);
7  Agent a = new Agent();
8  a.setCaseBase(cb3);
```

Initially a previously created case base is loaded (line 1) and a feature selection algorithm is chosen (line 2). This feature selection algorithm then analyzes the case base (line 3). The resulting case base from this preprocessing step ($cb2$) will be identical to the input case base ($cb1$) but the algorithm will have calculated the optimum binary weight for each type of sensory input and those weights can be retrieved (line 4). Those weights can be used to create a second preprocessing step (line 5) and that preprocessing step can be applied to the case base (line 6). This preprocessing step will modify the input case base ($cb2$) by removing any features that were found to have a weight of 0 (or less). The output case base (*cb*3) will therefore be less than (or equal to) the size of the input case base. A new agent can be created (line 7) and set to use the preprocessed case base (line 8).

A soccer playing agent has a set $\mathcal{A}_{SOCCER}$ of three possible atomic actions: *kick* (A_{kick}), *dash* (A_{dash}) and *turn* (A_{turn}).

$$\begin{aligned} \mathcal{A}_{SOCCER} &= \{A_{kick}, A_{dash}, A_{turn}\} \\ A_{kick} &= < f_{power}, f_{direction} > \\ A_{dash} &= < f_{velocity} > \\ A_{turn} &= < f_{angle} > \end{aligned}$$

Unlike in the physical robotic domains, where the actions did not have parameters, the soccer actions all have associated parameter features. The *kick* action shows an example of an action that can have multiple parameters since it has both a kick power (f_{power}) and direction ($f_{direction}$).

The observing agent learnt from an expert that turns until it can see the soccer ball, runs toward the ball and kicks the ball toward the opponent's goal net. A case base of 3500 cases was used along with 5000 test cases. However, unlike the previously described experiments the case base was preprocessed using an approach similar to the one described above (calculating optimum feature weights and filtering irrelevant features). The agent was able to achieve a *kick* accuracy of approximately 72%, a *dash* accuracy of approximately 93% and a *turn* accuracy of approximately 96% (overall accuracy of 87%). Without performing preprocessing, the overall accuracy of the agent is approximately 71% (the *kick* accuracy is particularly bad at 22%).

C. Tetris

Our final case study will involve the game of Tetris. In Tetris, there is a rectangular game region where the player stacks incoming game pieces of varying shapes. When the pieces form a horizonal line from one side of the game region to the other, the entire line is removed from the game region thereby freeing space. The player must strategically place pieces in order to ensure the stacked pieces do not reach the top of the game region.

The sensory input in Tetris S_{TETRIS} contains two sub-inputs: the current state of the game region (S_{region}) and the current piece that needs to be placed (S_{piece}).

$$S_{TETRIS} =< S_{region}, S_{piece} >$$

Each of these sub-inputs is represented by a matrix containing information about which squares are currently occupied. The game region is a 20×10 rectangle that contains 200 squares and the piece is a 4×4 rectangle that is composed of 16 squares. Each square S_{cell} contains a feature ($f_{occupied}$) representing if the square is occupied or not.

$$\begin{aligned} S_{region} &= < S_{cell}^{1}, \ldots, S_{cell}^{200} > \\ S_{piece} &= < S_{cell}^{1}, \ldots, S_{cell}^{16} > \\ S_{cell} &= < f_{occupied} > \end{aligned}$$

Unlike the soccer domain, where there was partial observability, the Tetris agent is always able to see the entire game region and game piece. Due to the full observability of these inputs it was possible to model them as being ordered and of a fixed-length (length 200 and 16 respectively) rather than as multi-valued inputs with a time-varying number of elements. This is important because similarity calculations become more computationally expensive with multi-valued inputs [10] and should therefore be avoided, if possible, when the agent has real-time constraints. Here, the similarity strategy of the complex inputs just calculates the mean of the similarities of the atomic inputs (like was done in the robot case study). The designer could also create a custom similarity strategy that takes inspiration from human players [7].

In Tetris, there is only one action to perform. However, unlike the previous domains, this action $A_{movepiece}$ is a complex action that contains two atomic actions. The first sub-action A_{slide} is related to how many squares the game piece should be moved horizontally (with positive values representing sliding right and negative left) and the other A_{rotate} is related to how many times the game piece should be rotated by 90 degrees clockwise.

$$\begin{aligned} \mathcal{A}_{TETRIS} &= \{A_{movepiece}\} \\ A_{movepiece} &= < A_{slide}, A_{rotate} > \\ A_{slide} &= < f_{slide} > \\ A_{rotate} &= < f_{rotate} > \end{aligned}$$

Since $A_{movepiece}$ is a complex action, both of the sub-actions will be performed. If the player did not want to perform one, or both, of the sub-actions they could set their parameters, f_{slide} or f_{rotate}, to be zero.

In our experiments, the observing agent learnt by watching a Tetris player and generated $100,000$ cases for the case base. When playing Tetris by itself, the agent was able to complete an average of approximately 2 lines per game (over 350 test games). While this performance is far worse than the expert, it does show that the agent is able to reproduce the behaviour of the expert (completing lines in Tetris) to some extent. Comparatively, when the agent just placed the pieces at random, it completely an average of approximately 0.1 lines per game (over 350 games). One reason for the poor performance of the agent is the large state-space of Tetris (approximately 2^{216} using our representation). This is compounded by the fact that the expert plays near-optimally and therefore rarely puts itself in disadvantageous positions. However, the learning agent will make mistakes which can cause the environment state to be significantly different from any it has encountered during observation. Active case acquisition approaches [11], [12] can be used to combat this and ensure no areas of the problem-space are underrepresented in the case base.

VI. Related Work

The majority of learning by observation systems, including those using case-based reasoning as well as other reasoning approaches, are developed to learn a specific behaviour or operate in a single domain. Some of these approaches use heuristics that contain information about the expert's behaviour [15], [7], contain hard-coded models of the expert [16] or access information about the expert's internal reasoning that would not be available through observation [17]. Others use inputs that ignore external stimuli [18], contain meta-information related to the observed task [5], [4], [19], [8] or ignore multi-valued features [20]. While these systems work well in their designed domains, they can not be directly used in new domains or to learn different tasks.

The Darmok learning engine [9] uses case-based planning to perform learning by observation. Through the use of the MakeME PlayME middleware [21], Darkmok is able to learn in a variety of real-time strategy and interactive drama domains [22]. However, when generating cases, Darmok requires the goals of the expert to be defined in order to create plans. This requires explicit knowledge about the behaviour of the expert being observed and would need to be modified if the behaviour of the expert changed. Also, Darmok has difficulty learning reactive behaviours [23]. To our knowledge, Darmok is the only other domain-independent learning by observation system.

Several other frameworks for case-based reasoning applications exist, with jCOLIBRI [24] and myCBR [25] currently being the most actively developed. jCOLIBRI is a general-purpose, feature-rich CBR framework that allows the development of a wide variety of CBR applications. Like our framework, jCOLIBRI attempts to separate algorithms from the domain model. Another tool, myCBR, allows for rapidly prototyping CBR applications. Unlike jCOLIBRI, myCBR focuses on similarity based retrieval and does not place as much emphasis on other parts of the case-based reasoning cycle. The primary difference between these two frameworks and our framework is the scope. Both myCBR and jCOLIBRI look to provide general frameworks that can be applied in a variety of CBR areas such as textual CBR, recommender systems or conversational CBR systems. Our framework has been developed exclusively for learning by observation systems and has been optimized accordingly. We place a strong emphasis on the properties that agents will encounter in their environment like partial observability, non-determinism, complex environments and real-time constrains. Similarly, their frameworks are designed to be used on desktop computers or as web-based applications whereas ours is designed to be used on embedded systems.

VII. Conclusions

In this paper we have described a framework, jLOAF[1], for developing case-based reasoning agents that learn by observation. We looked to design a framework that did not require any knowledge about the behaviour or goals of the expert being observed. Instead, a learning agent only needs a definition of its input and action models along with interfaces to allow it to properly observe and interact with the environment. By separating the core reasoning algorithms (described in more detail in [10]) from any domain specific knowledge, the same reasoning system can be utilized in a variety of environments. Our framework design has also focused on addressing many of the properties of realistic environments like complexity, non-determinism, partial observability and real-time constraints.

While we have focused on a domain independent framework that is not biased toward any specific task, there is nothing limiting an agent designer from introducing such bias in order to optimize performance. For example, the sensory input models could be designed to only contain inputs that the expert uses during reasoning or use similarity metrics that are tailored to a specific behaviour. Ideally, our framework looks to learn such optimization information during the preprocessing steps but it could also be hard-coded in order to save time.

We presented four case studies, in both simulated and physical environments, that show how inputs and actions can be modelled using our framework. Our examples included domains with simple sensor systems, high-level object detection, partial observability and full observability. Additionally, we described domains with parameter-free actions, multi-parameter actions and complex sequences of actions. Our experimental evaluation shows that the same agent, without changing the reasoning module, can successfully learn a variety of behaviours in several different domains. Not only do these agents perform well in experimental evaluations but they are also able to perform the learnt behaviour when placed in the environment[2].

There are several limitations of this framework. If an agent is only going to be used for a single task and domain knowledge can be obtained inexpensively, using our framework may not be appropriate. The preprocessing approaches would likely not learn the domain knowledge as precisely as if a domain expert provided it. Additionally, since no information about the goals or behaviours of the

[1] A reference implementation, that has been used in all described domains, is available: http://www.nmai.ca

[2] 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/

expert are provided the agent can not make use of techniques like reinforcement learning since it does not know when it has done something correctly or incorrectly. Like all learning by observation systems, the agent's performance is heavily influenced by the quality of observations (both the problem-space coverage and amount of noise). Our framework attempts to deal with these issues thought case acquisition techniques but they still remain difficulties. Our future work will look to further examine techniques for case acquisition, especially those that are able to minimize noisy or erroneous observations. We will also examine how the sensor and action models can be built dynamically so that sensors can be easily added or removed from a robot at runtime.

REFERENCES

[1] M. Wooldridge, *An introduction to multiagent systems*. John Wiley and Sons, 2002.

[2] A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches," *AI Communications*, vol. 7, no. 1, pp. 39–59, 1994.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addision-Wesley, 1995.

[4] S. Flinter and M. T. Keane, "On the automatic generation of cases libraries by chunking chess games," in *1st International Conference on Case-Based Reasoning*, 1995, pp. 421–430.

[5] M. Fagan and P. Cunningham, "Case-based plan recognition in computer games," in *5th International Conference on Case-Based Reasoning*, 2003, pp. 161–170.

[6] M. W. Floyd, A. Davoust, and B. Esfandiari, "Considerations for real-time spatially-aware case-based reasoning: A case study in robotic soccer imitation," in *9th European Conference on Case-Based Reasoning*, 2008, pp. 195–209.

[7] H. Romdhane and L. Lamontagne, "Forgetting reinforced cases," in *9th European Conference on Case-Based Reasoning*, 2008, pp. 474–486.

[8] J. Rubin and I. Watson, "Similarity-based retrieval and solution re-use policies in the game of Texas Hold'em," in *18th International Conference on Case-Based Reasoning*, 2010, pp. 465–479.

[9] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *7th International Conference on Case-Based Reasoning*, 2007, pp. 164–178.

[10] M. W. Floyd, B. Esfandiari, and K. Lam, "A case-based reasoning approach to imitating RoboCup players," in *21st International Florida Artificial Intelligence Research Society Conference*, 2008, pp. 251–256.

[11] M. W. Floyd and B. Esfandiari, "An active approach to automatic case generation," in *8th International Conference on Case-Based Reasoning*, 2009, pp. 150–164.

[12] ——, "Supplemental case acquisition using mixed-initiative control," in *Twenty-Fourth International Florida Artificial Intelligence Research Society Conference*, 2011, pp. 395–400.

[13] E. L. Rissland, D. B. Skalak, and M. T. Friedman, "Case retrieval through multiple indexing and heuristic search," in *13th International Joint Conference on Artificial Intelligence*, 1993, pp. 902–908.

[14] RoboCup, "Robocup official site," http://www.robocup.org, 2011. [Online]. Available: http://www.robocup.org

[15] J. Dinerstein, P. K. Egbert, D. Ventura, and M. Goodrich, "Demonstration-based behavior programming for embodied virtual agents," *Computational Intelligence*, vol. 24, no. 4, pp. 235–256, 2008.

[16] C. G. Atkeson and S. Schaal, "Robot learning from demonstration," in *Fourteenth International Conference on Machine Learning*, 1997, pp. 12–20.

[17] D. H. Grollman and O. C. Jenkins, "Learning robot soccer skills from demonstration," in *IEEE International Conference on Development and Learning*, 2007, pp. 276–281.

[18] A. Coates, P. Abbeel, and A. Y. Ng, "Learning for control from multiple demonstrations," in *25th International Conference on Machine Learning*, 2008, pp. 144–151.

[19] K. Gillespie, J. Karneeb, S. Lee-Urban, and H. Muñoz-Avila, "Imitating inscrutable enemies: Learning from stochastic policy observation, retrieval and reuse," in *18th International Conference on Case-Based Reasoning*, 2010, pp. 126–140.

[20] C. Thurau and C. Bauckhage, "Combining self organizing maps and multilayer perceptrons to learn bot-behavior for a commercial game," in *GAME-ON Conference*, 2003.

[21] P. P. Gómez-Martín, D. Llansó, M. A. Gómez-Martín, S. Ontañón, and A. Ram, "MMPM: A generic platform for case-based planning research," in *Workshop on Case-Based Reasoning for Computer Games at the 18th International Conference on Case-Based Reasoning*, 2010, pp. 45–54.

[22] M. Mehta, S. Ontañón, T. Amundsen, and A. Ram, "Authoring behaviors for games using learning from demonstration," in *Workshop on Case-Based Reasoning for Computer Games at the 8th International Conference on Case-Based Reasoning*, 2009.

[23] S. Ontañón and A. Ram, "Case-based reasoning and user-generated AI for real-time strategy games," in *Artificial Intelligence for Computer Games*, P. A. Gonzáles-Calero and M. A. Gomez-Martín, Eds., 2011, pp. 103–124.

[24] B. Díaz-Agudo, P. A. González-Calero, J. A. Recio-García, and A. A. Sánchez-Ruiz-Granados, "Building CBR systems with jCOLIBRI," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 68–75, 2007.

[25] A. Stahl and T. Roth-Berghofer, "Rapid prototyping of CBR applications with the open source tool myCBR," in *9th European Conference on Case-Based Reasoning*, 2008, pp. 615–629.